Machine Learning Projects for .NET Developers

Apress®

机器学习项目开发实战

[美] Mathias Brandewinder 著
姚军 译

- 来自.NET专家的声音
- 从数据中学习，让应用更“聪明”

中国工信出版集团
人民邮电出版社
POSTS & TELECOM PRESS

Machine Learning Projects for .NET Developers

机器学习项目开发实战

[美] Mathias Brandewinder 著
姚军 译

人民邮电出版社
北京

图书在版编目（CIP）数据

机器学习项目开发实战 /（美）马蒂亚斯·布兰德温德尔（Mathias Brandewinder）著 ；姚军译. -- 北京 ：人民邮电出版社，2016.8（2016.11重印）
ISBN 978-7-115-42951-3

Ⅰ. ①机… Ⅱ. ①马… ②姚… Ⅲ. ①机器学习 Ⅳ. ①TP181

中国版本图书馆CIP数据核字(2016)第162289号

版 权 声 明

◆ 著　[美] Mathias Brandewinder
译　姚　军
责任编辑　王峰松
责任印制　焦志炜

◆ 人民邮电出版社出版发行　北京市丰台区成寿寺路 11 号
邮编　100164　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京九州迅驰传媒文化有限公司印刷

◆ 开本：800×1000　1/16
印张：17.5
字数：382 千字　2016 年 8 月第 1 版
印数：2 501–3 100 册　2016 年 11 月北京第 2 次印刷

著作权合同登记号　图字：01-2016-4639 号

定价：59.00 元

读者服务热线：(010) 81055410　印装质量热线：(010) 81055316
反盗版热线：(010) 81055315

内容提要

本书通过一系列有趣的实例，由浅入深地介绍了机器学习这一炙手可热的新领域，并且详细介绍了适合机器学习开发的 Microsoft F#语言和函数式编程，引领读者深入了解机器学习的基本概念、核心思想和常用算法。书中的例子既通俗易懂，同时又十分实用，可以作为许多开发问题的起点。通过对本书的阅读，读者无须接触枯燥的数学知识，便可快速上手，为日后的开发工作打下坚实的基础。本书适合对机器学习感兴趣的.NET 开发人员阅读，也适合其他读者作为机器学习的入门参考书。

关于作者

Mathias Brandewinder 是 Microsoft F# 最有价值专家（MVP），住在加州旧金山，在那里他为 Clear Lines Consulting 工作。作为一名当之无愧的数学极客，他很早就对构建模型帮助其他人利用数据做出更好的决策感兴趣。他拥有商业、经济和运营研究等多个硕士学位，在到达硅谷之后不久便爱上了编程。从.NET 刚出现时开始，他就专业开发软件，为各行各业开发业务应用程序，重点是预测模型和风险分析程序。

关于译者

姚军，曾在多家券商任IT经理，在系统集成、数据库、网络系统方面有近20年经验，主导及参与了多个大型系统集成项目的需求分析、实施及维护。由于工作原因，在计算机领域涉猎极广，自2006年开始，工作之余将大量精力投入IT图书的翻译及编著工作，曾参与“全国网络技术水平考试丛书”的编写工作，译作甚丰，如《Python金融大数据分析》《数据驱动的网络分析》《软件架构师的12项修炼：技术技能篇》《增量承诺螺旋模型：系统和软件开发的成功之道》《VMware vCAT权威指南：成功构建云环境的核心技术和方法》等。

关于技术评审

Scott Wlaschin 是一位.NET 开发人员、架构师和作家。他在不同的领域有 20 多年的经验，从高级 UX/UI 到低级数据库实现均有涉猎。他曾经用多种语言编写重要的代码，其中最喜欢的是 Smalltalk、Python 和更新的 F#，他在 fsharpforfunandprofit.com 上发表关于 F# 的博客。

致谢

感谢我的父母，我在充满书香的家庭中长大，书籍对今天的我产生了深远的影响。我对它们的热爱是从事这个疯狂项目的部分原因，尽管许多人警告我，这一旅程充满艰辛，我仍努力地尝试写一本自己的书。旅途的艰辛是值得的，对此我满怀骄傲：我也能写一本书了！为此（还有许多其他方面），我必须感谢父母。

孤独的旅程毫无趣味，我很幸运有 3 位出色的同行者：无畏的 Gwenan、智慧的 Scott 和坚定的 Petar。Gwenan Spearing 和 Scott Wlaschin 认真地审核手稿，为我提供了宝贵的反馈意见，保证这个项目处于正确的方向。有了他们，最终的结果大不相同。书中最好的部分源自他们的贡献，而读者找到的任何问题都应归咎于我！我要衷心地感谢 Petar Vucetin，和他成为商业合作伙伴和朋友是一种幸运。他在我情绪不佳的时候首当其冲，在这种时候仍然鼓励我，为我提供完成任务所需的时间和空间。感谢你，伙计！——你是真正的朋友。

本书写作期间还有许多人帮助过我，无法一一列举。感谢帮助我实现本书，提供代码、建议或者溢美之辞的人们——你们知道我指的是谁！特别要感谢 F#社区。这是一个坦率的社区（显然有时候这令人有些烦恼），但更重要的是，认识社区的这么多人是快乐和灵感的源泉。请保持这种令人敬畏的状态！

当然，没有燃料就走不了长路。这段特别旅程的动力是咖啡因，在我看来，旧金山的咖啡馆有最好的玛琪雅朵咖啡，让我在过去的一年半中每天都有好的开始。

前言

如果你手里拿着这本书，我就可以认定你是对机器学习感兴趣的.NET 开发人员了。你可能对编写 C#应用程序很熟悉，开发的很有可能是业务线应用程序。以前你可能遇到过 F#，也可能没有。而且，你很有可能对机器学习感到好奇。这一主题每天都见诸报端，因为它和软件工程有着很紧密的联系，但是使用的是不熟悉、看似有些抽象的数学概念。简而言之，机器学习看上去是有趣的主题、值得学习的实用技能，但是从哪里入手难以说清。

本书的意图是作为开发人员的机器学习入门书。我的主要目标是使熟悉代码编写的读者（而不是数学家）容易理解书中的主题。对数学的喜爱当然没有坏处，但是本书通过实用的示例学习核心概念，说明其中的工作原理。

什么是机器学习？机器学习是一种编程艺术，所编写的计算机程序随着可用数据越来越多而更好地执行任务，无须开发人员更改代码。

上述定义相当宽泛，反映了机器学习广泛适用于各个领域这一事实。但是，该定义中的一些具体特征值得更详细说明。机器学习是关于程序编写的学科，这些代码运行于生产环境并执行某项任务，这使它不同于统计学。机器学习是一个跨学科的领域，这个主题既和倾向于数学的研究人员相关，也和软件工程师相关。

定义中另一个有趣的部分是数据。机器学习是关于利用可用数据解决实际问题的学科。使用数据是机器学习的关键部分，理解数据、研究如何从中提取有用信息，往往比使用的特定算法更重要。因此，我们将从数据开始了解机器学习。每章都从一个真实的数据集和所要解决的特定问题开始，数据中包含了现实世界中的所有不完善和意外。由此，我们将在这一背景下从头开始构建问题解决方案，在需要的时候介绍思路。在此过程中，我们将创建一个基础，帮助你理解不同思路的组合使用，使你在以后需要的时候更有效率地使用库或者框架。

我们的探索从熟悉的 C#和 Visual Studio 开始，但是在取得进展之后将介绍 F#，这是一种特别适合于机器学习问题的.NET 语言。正如机器学习，函数式编程一开始令人生畏。然而，一旦掌握了诀窍，F#就会变得很简单且极具效率。如果你完全是 F#的初学者，本书将告诉你该语言所需了解的一切，你将在现实、有趣的问题中学习如何高效地使用该语言。

学习过程中，我们将探索各种各样的问题，帮助你理解机器学习能使应用程序变得更好的领域，有些方法可能出人意料。我们将探索图像识别、垃圾邮件过滤器和自我学习游戏以及其他一些问题。而且，在我们共同的旅途上，你将发现机器学习并没有那么复杂，相当简单的模型就能产生令人惊讶的出色结果。最后，你将会发现，机器学习非常有趣！好了，不多啰唆了，让我们一起对付第一个机器学习问题吧！

目录

第 1 章

256 级灰度

构建自动识别数字图像的程序

如果你打算建立一个当前技术热点的列表，机器学习当然会名列前茅。然而，虽然这个术语到处出现，但是它的真实含义往往含混不清。它是和“大数据”或者“数据科学”一样的东西吗？它和统计学有何不同之处？表面上，机器学习似乎是一种奇特、令人畏惧的专业，使用令人眼花缭乱的数学知识和算法，和软件工程师的日常活动没有多少共同之处。

在本章以及本书余下的部分中，我的目标是和大家一起完成实际项目，以此阐明机器学习的原理。我们将循序渐进地解决问题，主要是从头开始编写代码。通过讲述这种方法，我们可以理解工作原理的细节，逐步说明广泛适用的核心思路和方法，并帮助你为以后构建专用程序库打下坚实的基础。在第 1 章中，我们将深入探讨一个经典问题——手写数字识别，同时完成以下几件工作：

- 建立适用于大部分机器学习问题的方法论。机器学习模型的开发与标准业务线应用程序有微妙的不同，将带来特殊的挑战。学到本章的最后，你将会理解交叉验证的概念、重要性以及使用方法。
- 帮助你理解如何“考虑机器学习”，以及如何看待机器学习问题。我们将讨论相似性和距离之类的思路，这些思路是大部分算法的核心。我们还将说明，虽然数学是机器学习的重要组成部分，但是这个方面可能被过分强调了，有些核心思路实际上相当简单。我们将从比较简单的算法开始，你会看到，这些算法实际上工作得很好！
- 了解如何用 C#和 F#解决问题。我们将从实现 C#解决方案开始，然后提供 F#的等价解决方案。F#是一种特别适合于机器学习和数据科学的.NET 语言。

在第 1 章就碰上这样的问题，似乎会令人畏缩——但是不要被吓住！从表面上看这个问题很难，但是你将会发现，我们仅用相当简单的方法，就能够创建相当有效的解决方案。再说，解决小儿科的问题有什么意思？

1.1　什么是机器学习

什么是机器学习？机器学习的核心是编写能够学习如何根据经验执行任务的程序，而不需要明确地编程说明如何解决问题。这仍然是一个模糊的定义，产生了这样的问题：如何精确地定义“学习”？下面是一个有些乏味的定义。

如果一个程序得到更多的数据点，它就自然能够更好地执行指定的任务，那么我们就称它是“学习”。也可以反过来看这个定义：如果你不顾观察到的结果，一次又一次地重复进行相同的工作，那当然就没有“学习”。

上述定义很好地总结了“进行机器学习”的含义。你的目标是编写一个自动执行某些任务的程序。这个程序应该能够从经验中学习，经验的形式可能是预先存在的过往观测数据集，也可能是程序本身执行任务时积累的数据（称作“在线学习”）。随着可用数据越来越多，无须更改程序代码本身，程序就应该变得更加擅长此项任务。

编写这样一个程序时，你的任务包括两个方面。首先，你的程序需要可从中学习的数据。机器学习的一个重要组成部分是收集数据，并准备数据使其形式可供程序使用。将原始数据重新组织为更好地表现问题域、程序可理解格式的过程称作特征抽取。此后，程序必须能够理解自己执行任务的质量，以便根据经验调整和学习。因此，定义一个指标，准确地计量任务完成质量，是至关重要的。

最后，机器学习需要一些耐心、好奇心和许多创意！你需要选择一种算法，为它提供数据以训练某种预测模型，验证模型的表现，还需要改进和迭代——可能是定义新功能，也可能是选择新的算法。

这一周期——从训练数据中学习，评估验证数据并改进——是机器学习过程的核心。这也是真正起作用的科学方法：你努力地找出一种模型，通过公式化各种假设，进行一系列验证试验确定如何推进，准确地预测世界的发展。

在进入我们的第一个问题之前，必须简要地说明两点。首先，这只是对机器学习的宽泛描述。机器学习适用于多个领域的问题，从检测垃圾邮件和自动驾驶汽车，到推荐你可能喜欢的电影，自动翻译或者使用医疗数据帮助诊断。虽然只有深入理解各个领域的专业知识及需求才能成功地应用机器学习技术，但是原理和方法大体相同。

其次，请注意我们的机器学习定义中明确地提到了“编写程序”，和主要考虑验证模型是否正确的统计学不同，机器学习的最终目标是创建一个在生产环境中运行的程序。这本身就使之成为一个有趣的领域，因为本质上机器学习是跨学科的（在统计方法和软件工程上都成为专家是很难的），它也为软件工程师开创了一个非常激动人心的新领域。

现在我们已经有了基本的定义，下面进入第一个问题。

1.2 经典的机器学习问题：图像分类

图像（特别是笔迹）识别是机器学习的一个经典问题。首先，这个问题有极其有益的应用。通过自动识别书信上的地址或者邮政编码，邮局就可以有效地分拣信件，免于人工进行这一乏味的工作；ATM 机器如果能够识别金额，就可以在机器上存款，加快资金入账的速度，减少在银行排队的需求。想象一下，如果所有人类手写的文档都能够数字化，搜索和研究信息该有多么容易！其次，这个问题很难：人类的笔迹（即使是印刷体）有各种各样的变化（大小、形状、倾斜），人们可以毫无问题地识别不同人写的字母和数字，计算机处理起来却十分困难。这就是断定某人是真人还是计算机时 CAPTCHA 如此简单有效的原因。人类的大脑在识别字母和数字的能力上有着可怕的能力，即使这些图像严重失真也不在话下。

有趣的事实：CAPTCHA 和 reCAPTCHA

CAPTCHA（Completely Automated Public Turing test to tell Computers and Humans Apart，全自动区分计算机和人类的图灵测试）是设计用于区分人和机器人的一种机制。为了确保用户是一个真正的人，CAPTCHA 显示一段故意打乱的文本，使自动化电脑程序难以分辨。更有趣的是，这一思路已经扩展为 reCAPTCHA。reCAPTCHA 显示两个图像而不仅仅是一个：其中一个用于过滤机器人，另一个是真正数字化的文本（见图 1-1）。每当人们这样登录时，它还有助于存档文件的数字化，例如《纽约时报》的过刊，每次数字化一个词。

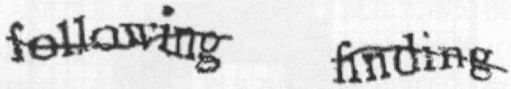


图 1-1 reCAPTCHA 示例

1.2.1 挑战：构建一个数字识别程序

我们所要解决的问题是“数字识别程序”，这个问题是直接从 Kaggle.com 机器学习竞赛中借用的。可以在网站上找到所有相关信息：http://www.kaggle.com/c/digit-recognizer。

下面就是这道难题：我们有 50000 幅图像的数据集。每个图像是由一个人写下的单个数字，以 28×28 像素分辨率扫描，编码为灰度图像，每个像素使用 256 个可能的灰度中的一个（从全白到全黑）。我们知道每次扫描的正确答案，也就是那个人写下的数字。这一数据集被称作“**训练集**”。现在，我们的目标是编写一个程序，从训练集中学习，并使用该信息预测之前从未见过的图像：该图像是 0，还是 1，还是……

从技术上说，这是所谓的“**分类**”问题：目标是将图像分到已知的“类别”中，这就是“分类”的由来。在本例中，我们有 10 个类别，代表 0～9 的数字。机器学习的不同风格取决于所要解决的问题类型，分类只是其中之一，但可能是最有代表性的问题。我们将在本书介绍更多此类问题！

那么，如何解决这个问题？我们先从一个不同的问题开始，想象一下我们只有两个图像：0 和 1（见图 1-2）。

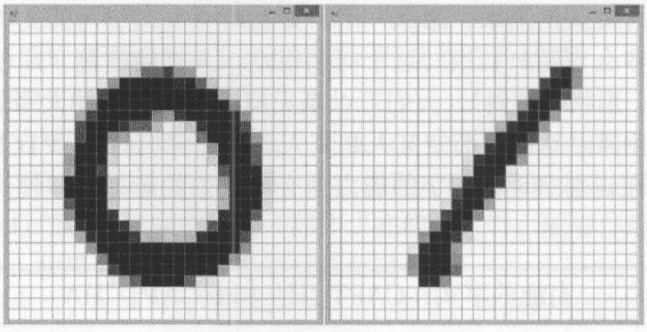


图 1-2　数字化的 0 和 1 样本

假定现在我为你提供图 1-3 中的图像，并提出下列问题：图 1-2 中显示的两个图像中，哪个与图 1-3 最为相像？

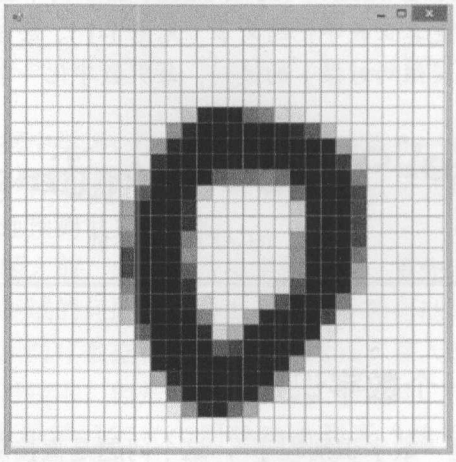


图 1-3　需要分类的未知图像

作为人，我想你一定觉得这个问题太简单了：“很明显，是第一个。”就这一点而言，我想两岁大的孩子都会觉得很简单。真正的问题是，如何将大脑所变的“魔术”翻译成代码？

解决这个问题的方法之一是调换一下问题的措辞：最相似的图像是**差别最小**的图像。在这个框架下，可以开始“找出差异”，逐个像素比较两个图像。图 1-4 中的图像显示了差异的“热图”：两个像素差别越大，颜色就越深。

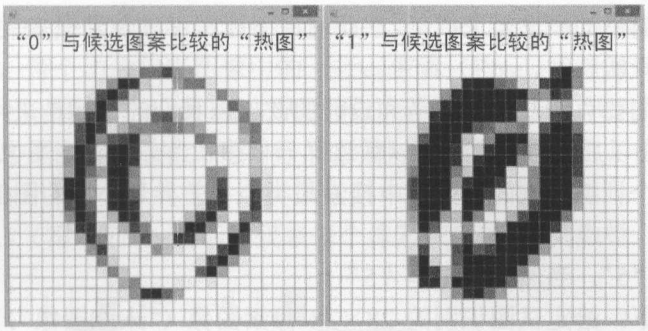


图 1-4 突出图 1-2 和图 1-3 不同之处的“热图”

在我们的例子中，这种方法似乎很有效。第二个图像“大不相同”，中间有一个很大的黑色区域，而第一个图像（画出了两个“0”之间的差异）大部分是白色，只有稀少的深色区域。

1.2.2 机器学习中的距离函数

现在我们可以加总各个像素的差异，用单一数字总结两个图像的差异。这样，对于相似的图像可以得到一个较小的数，对于不相似的图像则得到较大的数。在此，我们可以定义两个图像的“**距离**”，以描述其接近的程度。绝对完全相同的两个图像距离为 0，不同的像素越多，距离就越大。另一方面，我们知道距离为 0 的含义是完全匹配，也知道这是我们所希望出现的最佳状况。但是，我们的相似度计量是有局限性的。例如，如果你简单地复制一个图像，但是向左移动一个像素，逐个像素的距离可能会变得相当大，甚至是在图像本质上完全相同的情况下。

距离的概念在机器学习中很重要，以某种形式出现在大部分模型中。距离函数是将你所要实现的功能翻译为机器能够使用的形式的手段。通过减小事物（如两个图像）的复杂度，形成单一数字，算法就可能产生作用——在本例中，算法可以决定两个图像是否相似。与此同时，将复杂度减小为单一数字也会带来某些细节在“翻译中丢失”的风险，就像上述的图像移动的情况。

距离函数还常常以另一个名称出现在机器学习中：**代价函数**。两者本质上相同，只是从不同的角度看待问题。例如，如果我们试图预测一个数字，预测的误差——也就是预测值与实际值的差别——就是距离。但是，同样可以用代价描述这一情况：较大误差的“代价很高”，对模型的改进能够降低其代价。

1.2.3 从简单的方法入手

现在，我们暂时忽略上述问题，遵循一种在编写软件和开发预测模型中都很奇妙的方法，继续解决问题——最简单的方法能不能生效？先从简单的方法入手，看看会发生什么情况。如果这种方法有效，就没有必要采用复杂的方法，从而更快地解决问题。如果无效，你可以

花费少量时间建立简单的概念验证模型，在此过程中通常可以学到很多关于问题空间的知识。无论如何，这都是一次胜利。

因此，现在我们要克制住过度思考和设计的欲望，我们的目标是采用自己认为可能成功的最简单方法，并在以后改进。我们可以做这样一件事：当必须确定图像所代表的的数字时，可以在包含 50000 个训练示例的已知库中搜索最类似（差异最小）的图像，并预测图像所代表的数字。

如果看上去像“5”，当然就肯定是“5”！

算法的轮廓如下：给定试图识别的 28×28 像素图像（“未知图像”）和 50000 个训练示例（28×28 像素图像和一个标签），我们将：

- 计算未知图像和每个训练示例的总差值。
- 找出差值最小的训练示例（“最接近者”）。
- 预测“未知”与“最接近者”相同。

让我们开始动手吧！

1.3　我们的第一个模型（C#版本）

我们从 C#实现开始热身（这应该是读者熟悉的领域），并在 Visual Studio 中创建一个 C#控制台应用程序。我将自己的解决方案称作 DigitsRecognizer（数字识别器），C#控制台应用名为 CSharp——尽管取些比我更有创意的名称！

1.3.1　数据集组织

显然，首先我们需要数据。我们从 http://1drv.ms/1sDThtz 下载 trainingsample.csv 数据集，保存在机器上的某个位置。同一个位置还有第二个文件—— validationsample.csv，这个文件以后才会用到，但是我们现在就获取它。这两个文件采用 CSV 格式（逗号分隔值），结构如图 1-5 所示。第一行是文件头，后面的每行代表一个图像。第一列（“label”）表示图像代表的数字，接下来的 784 列（“pixel0”，“pixel11”）代表原始图像的每个像素，按照灰度编码（0～255，0 表示纯黑，255 表示纯白，中间的任何值是灰度级）。

```
1 label,pixel0,pixel1,pixel2,pixel3,pixel4,pixel5,pi
2 1,0,0,0
```

例如，第一行的数据表示数字 1，如果我们打算从这行数据重建真实的图像，可以将行分成 28 个“切片”，每个切片代表图像中的一行：像素（Pixel）0，像素 1…像素 27 编码图像的第一行，像素 28，像素 29…像素 55 代表第二行，依次类推。我们最终一共有 785 列数据：一列用于标签，其他 784 列代表 28 行×28 列=784 个像素。图 1-6 所示用简化的 4×4 图像描述了编码机制：

真实的图像是 1（第 1 列），然后是代表每个像素灰度的 16 列数据。

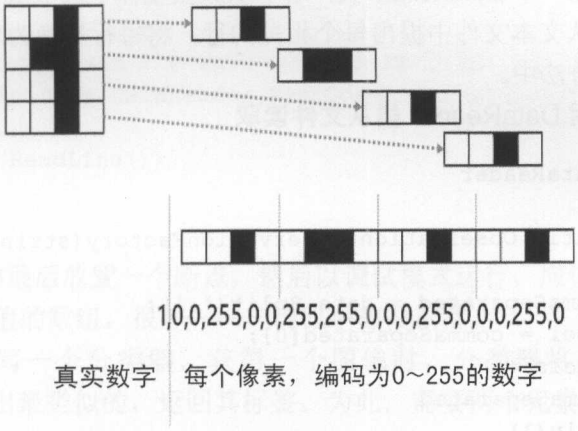


图 1-6 图像编码为 CSV 行（简化）

> **注意：**如果认真观察，就会注意到 trainingsample.csv 文件只有 5000 行，而不是前面提到的 50000 行。创建较小的文件是为了方便，只保留了原始数据最开始的一部分。50000 行这个数字并不算大，但是足以让我们的进展变得很慢，这令人不快，此时在较大的数据集上工作也没有什么价值。

1.3.2 读取数据

我们将用典型的 C#风格，围绕表示问题领域的几个类和接口构造代码。我们将在 Observation 类中保存每个图像的数据，用 IClassifier 接口表示算法，这样就可以在以后创建模型的变种。

第一步，我们需要从 CSV 文件读取数据，放入一组观测值中。下面我们进入解决方案，并在 CSharp 控制台项目中添加一个类，以保存观测值。

程序清单 1-1 在 Observation 类中保存数据

```
public class Observation
{
    public Observation(string label, int[] pixels)
    {
```

```
            this.Label = label;
            this.Pixels = pixels;
        }

        public string Label { get; private set; }
        public int[] Pixels { get; private set; }
    }
```

接下来，我们添加一个 DataReader 类，用于从数据文件中读取观测值。这里实际上是执行两个不同的任务：从文本文件中提出每个相关的行，将每行转换为观测值类型。我们将这些任务分别放在两个方法中。

程序清单 1-2　用 DataReader 类从文件读取

```
public class DataReader
{
    private static Observation ObservationFactory(string data)
    {
        var commaSeparated = data.Split(',');
        var label = commaSeparated[0];
        var pixels =
            commaSeparated
            .Skip(1)
            .Select(x => Convert.ToInt32(x))
            .ToArray();

        return new Observation(label, pixels);
    }

    public static Observation[] ReadObservations(string dataPath)
    {
        var data =
            File.ReadAllLines(dataPath)
            .Skip(1)
            .Select(ObservationFactory)
            .ToArray();

        return data;
    }
}
```

注意，我们的代码主要是 LINQ 表达式！面向表达式的代码（如 LINQ 或者后面将会看到的 F#）有助于编写非常清晰的代码，以直接的方式表达意图，通常比过程式编码有效得多。这种代码读起来更像自然语言："读取所有行，跳过文件头，根据逗号拆分各行，解析为整数，为我提供新的观测值"。如果我和同行交谈，这就是描述意图的方法，而这个意图很清晰地反映在代码中。这种代码特别适合于数据操纵任务，因为它提供了描述数据转换工作流的自然

手段，是机器学习的基础。毕竟，这就是 LINQ 的设计目的——“语言集成查询”！

我们已经有了数据、阅读器和保存它们的结构——下面在控制台应用中将其组合起来，用本地机器上实际数据文件的路径代替 trainingPath 中的 PATH-ON-YOUR-MACHINE。

程序清单 1-3 控制台应用程序

```
class Program
{
    static void Main(string[] args)
    {
        var trainingPath = @"PATH-ON-YOUR-MACHINE\trainingsample.csv";
        var training = DataReader.ReadObservations(trainingPath);

        Console.ReadLine();
    }
}
```

如果你在代码块的最后放置一个断点，然后以调试模式运行，应该看到 training 变量是一个包含 5000 个观测值的数组。很好——一切似乎都很正常。

下一个任务是编写一个分类器，获得一个图像时，分类器将其与数据集中的每个 Observation 比较，找出最类似的，返回其标签。为此，需要两个元素：Distance（距离）和 Classifier（分类器）。

1.3.3 计算图像之间的距离

我们从距离开始，所需的是一个方法，用以取得两个像素数组并返回描述它们的差异的数字。距离是算法中的易变领域，我们很可能想要试验不同的图像比较方法，找出最合适的方法。因此采用一种设计，使我们能够轻松地替换不同的距离定义且不需要做很多的代码更改，是最为可取的。接口提供了一种方便的机制，可以使用它避免紧密耦合，确保在以后想要更改距离代码时，不需要遇到令人烦恼的重构问题。所以，我们从一开始就提取一个接口。

程序清单 1-4 IDistance 接口

```
public interface IDistance
{
    double Between(int[] pixels1, int[] pixels2);
}
```

有了接口之后，需要一个实现。同样，我们现在将采用可能有效的最简单的方法。例如，如果我们想要的是计量两个图像的差异，为什么不逐个像素比较，计算每个像素的差值，然后加总其绝对值？完全相同的图像距离为 0，两个像素相差越远，两个图像的距离就越大。这种距离的名称为“**曼哈顿距离**”，实现起来相当简单，如程序清单 1-5 所示。

程序清单 1-5　计算图像之间的曼哈顿距离

```
public class ManhattanDistance : IDistance
{
    public double Between(int[] pixels1, int[] pixels2)
    {
        if (pixels1.Length != pixels2.Length)
        {
            throw new ArgumentException("Inconsistent image sizes.");
        }

        var length = pixels1.Length;

        var distance = 0;

        for (int i = 0; i < length; i++)
        {
            distance += Math.Abs(pixels1[i] - pixels2[i]);
        }

        return distance;
    }
}
```

有趣的事实：曼哈顿距离

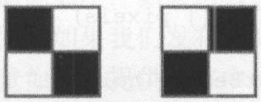
前面我已经提到过，距离可以用多种方法计算。我们在这里使用的具体公式被称作“曼哈顿距离”。这个名称来自于这样一个事实：如果你是纽约市的出租车司机，这就是计算两点之间驾车距离的方法。因为所有街道都被组织为一个完善的矩形网格，可以计算东/西位置和南/北位置之间的绝对距离，这就是我们的代码中所完成的工作。这种方法还有一种不那么有诗意的名称——L1 距离。

我们取两个图像并逐个像素比较，计算差值并返回总数，代表两个图像的距离。注意，这里使用的代码采用了过程式风格，完全没有使用 LINQ。实际上，我最初使用 LINQ 编写了这一段代码，但是老实说不喜欢结果的显示方式。在我看来，在某种程度（或者对于某些运算来说）上，以 C#语言编写的 LINQ 代码看起来有点过于复杂，这主要是因为 C#很冗长，尤其是其函数构造（Func<A,B,C>）。这也是比较两种风格的有趣示例。在此，要理解代码所做的是什么，需要逐行阅读并将其翻译为“人类的描述”。这段代码还使用了突变（Mutation）——这是一种需要注意的风格。

MATH.ABS()

你可能觉得奇怪，为什么我们要使用绝对值？为什么不简单地计算差值？为了理解这成

为问题的原因，考虑如下的例子：



如果我们使用像素颜色的“简单”差值，就会遇到一个微妙的问题。计算第一个和第二个图像的差值将得到-255+255-255+255=0，和第一个图像与自身的距离相同。这明显是不正确的。第一个图像明显和自身完全相同，而按照该指标，不同的图像 1 和 2 在外观上可能是相同的！使用绝对值的理由就是：如果不这么做，方向相反的差异就会相互抵消，结果是完全不同的图像可能会产生很高的相似度。绝对值保证不会有这种问题：任何差异都根据其量级进行处置，而不考虑其符号。

1.3.4 编写分类器

我们已经有了比较图像的方法，现在可以从一个通用接口开始，编写分类器了。在每种情况下，我们都预期使用一个两步的过程。提供一个已知观测值的训练集对分类器进行训练，一旦训练完成，我们就应该能够预测某个图像的标签了，见程序清单 1-6。

程序清单 1-6 IClassifier 接口

```
public interface IClassifier
{
    void Train(IEnumerable<Observation> trainingSet);
    string Predict(int[] pixels);
}
```

下面是实现前述算法的多种方法之一。

程序清单 1-7 基本分类器实现

```
public class BasicClassifier : IClassifier
{
    private IEnumerable<Observation> data;

    private readonly IDistance distance;

    public BasicClassifier(IDistance distance)
    {
        this.distance = distance;
    }

    public void Train(IEnumerable<Observation> trainingSet)
    {
        this.data = trainingSet;
```

```
    }

    public string Predict(int[] pixels)
    {
        Observation currentBest = null;
        var shortest = Double.MaxValue;

        foreach (Observation obs in this.data)
        {

            var dist = this.distance.Between(obs.Pixels, pixels);
            if (dist < shortest)
            {
                shortest = dist;
                currentBest = obs;
            }
        }

        return currentBest.Label;
    }
}
```

上述实现仍然是过程式的，但是应该不难理解。训练阶段简单地在分类器中保存训练观测值。为了预测图像所代表的数字，算法从训练集中搜索每个已知的观测值，计算与试图识别的图像之间的相似度，返回最接近匹配的图像标签。相当简单！

1.4　那么，如何知道程序有效？

我们拥有一个分类器——可以分类图像的出色代码，这很棒。

大功告成，可以交付程序了！等等，别忙！还有点小问题：我们对代码是否有效一无所知。作为软件工程师，知道“是否正常工作”是很简单的。拿起你的规格说明（每个人都有规格说明，不是吗？），编写测试（当然会这么做），运行测试，大功告成！如果出现问题，你就会知道。但是，我们关心的不是“程序正常工作”或者“程序出错”，而是“我们的模式是否擅长做出预测？”

1.4.1　交叉验证

交叉验证自然的出发点是计量模型执行任务的质量。在我们的例子中，这实际上很容易做到：为分类器提供图像，要求做出预测，与真正的答案比较，算出正确预测的数量。当然，为了做到这一点，我们必须知道正确答案。换言之，我们需要一个具有已知标签的图像数据集，用它测试模型的质量。这个数据集被称作**验证集**（有时候直接称作“测试数据”）。

你可能会问，为什么不使用训练集本身？我们可以训练分类器，然后在 5000 个例子上运行它。这不是一个很好的主意，原因如下：如果这么做，所计量的是模型学习训练集的质量。而你真正感兴趣的与此稍有不同：如果我们发布分类器，并开始向其提供之前从未遇见的新图像时，分类器的预期工作质量如何？提供训练中使用的图像可能做出乐观的估计。如果想要更实际的估计，就应该提供尚未使用的模型数据。

■ **注意**：让分类器使用训练集进行验证是可能大错特错的有趣示例。如果你尝试这么做，会发现每个单独的图像都可以正确识别，100%的正确率！对于这么简单的模型，结果似乎好得难以置信。真正发生的情况是：当算法在训练集中搜索最类似的图像时，每次都将找到完美的匹配，因为测试使用的图像本身就属于训练集。因此，当结果太好，不那么可信的时候，应该再次检查！

解决这种问题的通用方法称作**交叉验证**。将可用的数据拆分为训练集和验证集，留出部分数据。使用第一个数据集训练模型，第二个数据集用于评估模型的质量。

前面，你下载了两个文件 trainingsample.csv 和 validationsample.csv，我已经为你准备了两部分数据，无须重新进行。训练集是原始数据集 50000 个图像中的 5000 个样本，验证集是来自相同来源的另外 500 个图像。正如后面几章中将要看到的，进行交叉验证还有更有趣、但也需要更注意潜在陷阱的方法，不过，简单地将可用数据分解为两个单独的样本（如 80%/20%），是简单有效的入门手段。

1.4.2 评估模型质量

下面，我们编写一个类计算正确分类的比例，评估我们的模型（或者你希望尝试的其他模型）。

程序清单 1-8 评估基本分类器的质量

```
public class Evaluator
{
    public static double Correct(
        IEnumerable<Observation> validationSet,
        IClassifier classifier)
    {
        return validationSet
            .Select(obs => Score(obs, classifier))
            .Average();
    }

    private static double Score(
        Observation obs,
        IClassifier classifier)
    {
```

```
            if (classifier.Predict(obs.Pixels) == obs.Label)
                return 1.0;
            else
                return 0.0;
        }
    }
```

在此，我们使用了一个小技巧：向 Evaluator 传递一个 IClassifier 和一个数据集，对每个图像比较分类器预测值和正确值，为预测“计分”。如果匹配，记录“1”，否则记录“0”。使用数字代替真/假值，就可以求出代表正确率的平均值。

组合上述算法，看看我们的极简分类器在提供的验证数据集（validationsample.csv）上表现如何。

程序清单 1-9　训练和验证基本 C#分类器

```
class Program
{
    static void Main(string[] args)
    {
        var distance = new ManhattanDistance();
        var classifier = new BasicClassifier(distance);

        var trainingPath = @"PATH-ON-YOUR-MACHINE\trainingsample.csv";
        var training = DataReader.ReadObservations(trainingPath);
        classifier.Train(training);

        var validationPath = @"PATH-ON-YOUR-MACHINE\validationsample.csv";
        var validation = DataReader.ReadObservations(validationPath);

        var correct = Evaluator.Correct(validation, classifier);
        Console.WriteLine("Correctly classified: {0:P2}", correct);

        Console.ReadLine();
    }
}
```

现在运行这个程序，应该获得 93.4%的正确率，这可是在一个并不那么简单的问题上获得的成绩。我的意思是，我们能够以相当好的可靠性自动识别人类手写的数字！尤其是考虑到这是第一次尝试，我们有意地保持简洁性，这样的结果已经很不错了。

1.4.3　改进模型

那么，下一步是什么？我们的模型是不错，但是为什么要止步不前呢？毕竟，我们离 100%正确的理想还很远——能不能想出一些巧妙的改进措施，更好地做出预测？

这是验证集的关键之处。正如单元测试为你提供安全措施，在代码偏离轨道时提出警告一样，验证集为模型建立一个基准，使你不会陷入“盲飞”的境地。现在，可以自由地试验建模思路，通过验证就可以得出思考方向是否有前景的清晰信号。

在这一阶段，通常可以采用两种途径中的一种。如果模型足够好，可以收工——程序已经完成。否则，应该开始思考改善预测的方法，创建新模型，根据验证集运行模型，比较分类正确率，评估新模型是否工作得更好，逐步微调模型直到满意。

但是在开始试验改进模型的方法之前，现在似乎是介绍 F#的最佳时机。F#是一种绝妙的.NET 语言，很适合于机器学习和数据科学，它将使我们的模型试验变得更加简便。现在我们已经有了可以正常工作的 C#版本，下面让我们更深入一步，用 F#语言重写程序，以便比较两者之间的不同，更好地理解 F#的工作方式。

1.5 介绍用于机器学习的 F#

你是否注意到，运行我们的模型要花多少时间？为了了解模型的质量，在任何代码更改之后，都需要重建控制台应用并运行，重新加载数据，然后计算。步骤很多，如果数据集更大，一天当中的大多数时间就要用在等待数据加载上，这不是好的做法。

1.5.1 使用 F#交互执行进行实时脚本编写和数据研究

相比之下，F#自带一个非常方便的功能——Visual Studio 中的 F#交互执行（Interactive）。F#交互执行是一个 REPL（输入—求值—打印循环），本质上是一个实时脚本编写环境，可以在其中试验代码而无须经历前面描述的整个循环。

这样，我们将用一个脚本工作，而不是控制台应用程序。进入 Visual Studio 并在解决方案中添加一个新的库项目（见图 1-7），我们将其命名为 FSharp。

> **提示**：如果你用 Visual Studio 专业版或者更高版本开发，F#应该是默认安装的，其他情况请访问 F#软件基金会的网站 www.fsharp.org，网站上有全面的设置指南。

值得一提的是，你刚刚在包含现有 C#项目的.NET 解决方案中添加了一个 F#项目。F#和 C#完全可以互操作，可以毫无问题地相互通信——没有必要限制自己用一种语言解决所有问题。可惜，人们往往将 C#和 F#视为竞争语言，实际并非如此。它们可以很好地相互补充，两全其美：对于 C#最擅长的功能使用 C#，而在 F#最专精的地方则利用 F#!

在新项目中，应该看到一个名为 Library1.fs 的文件，这是.cs 文件在 F#中的等价物。但是，你有没有注意到名为 script.fsx 的文件？.fsx 文件是脚本文件，和.fs 文件不同，它们不是构建的一部分。它们可以用在 Visual Studio 之外，作为纯粹的独立脚本，这本身非常实用。在当前的背景（机器学习和数据科学中）下，我特别感兴趣的使用方法是在 Visual Studio

中，.fsx 文件组成了出色的“便笺本”，你不仅可以在其中试验代码，还能利用智能感知（IntelliSense）的所有好处。

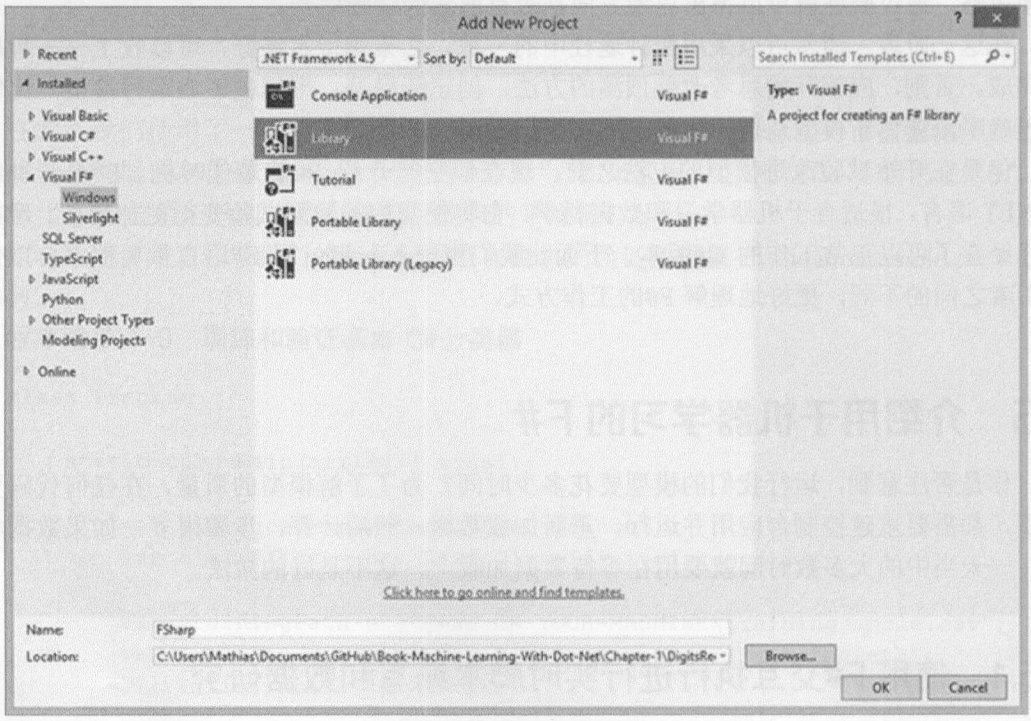


图 1-7　添加 F#库项目

进入 Script.fsx，删除其中的所有内容，在任意位置键入如下语句：

```
let x = 42
```

现在选择刚刚键入的代码行并单击鼠标右键，在上下文菜单中将看到“交互执行”（Execute In Interactive）选项，如图 1-8 所示。

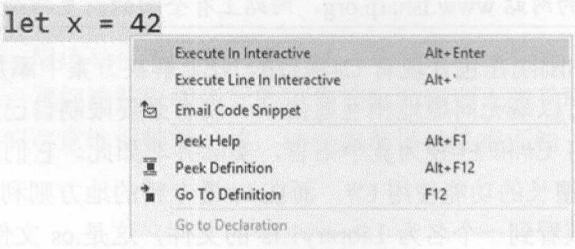


图 1-8　选择代码交互运行

继续——你应该看到标签为“F# Interactive”的窗口中出现结果（见图 1-9）。

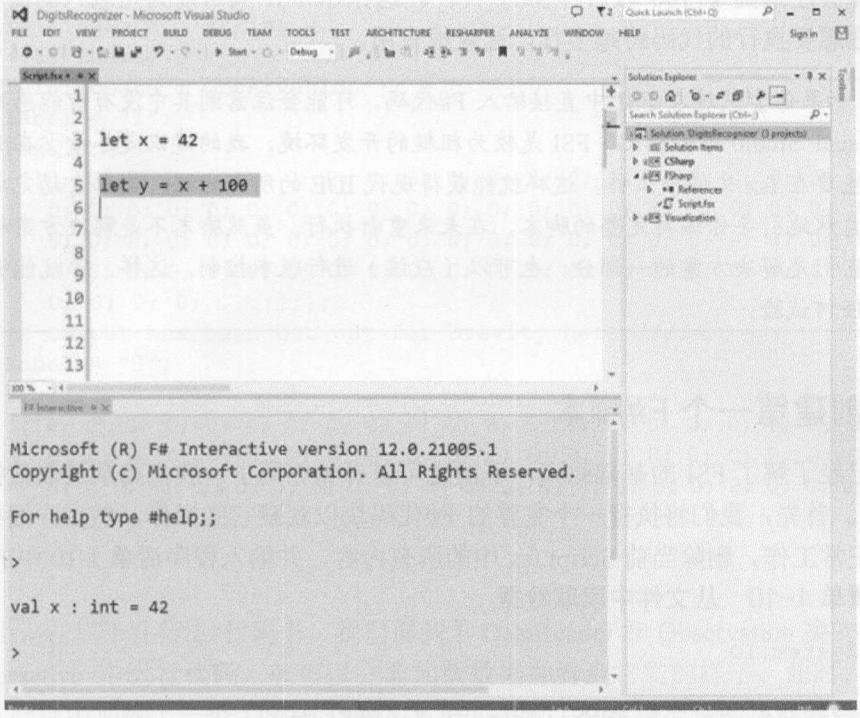


图 1-9 在 F# Interactive 中实时执行代码

提示：也可以使用快捷键 Alt+Enter 执行脚本文件中选中的任何代码。这比使用鼠标和上下文菜单快得多。对 ReSharper 用户有一个小小的警告：到目前为止，ReSharper 都有重置快捷键的“坏习惯”，所以如果使用 8.1 之前的版本，可能必须重新创建该快捷方式。

F# Interactive 窗口（大部分时候为了简洁我们称之为 FSI）以会话的形式运行。也就是说，在交互窗口中执行的任何命令都将保持在内存中，在用鼠标右键单击 F# Interactive 窗口并选择“重置交互会话”（Reset Interactive Session）之前都可以访问。

在本例中，我们简单地创建一个变量 *x*，值为 42。乍一看，这和 C#的 var x=42 语句类似，但两者之间有一些微妙的差别，我们将在后面讨论。现在，*x* “存在于” FSI 中，一直可以使用。例如，可以在 FSI 中直接输入：

```
> x + 100;;
val it : int = 142
>
```

FSI “记得” *x* 存在：不需要重新运行.fsx 文件中的代码，只要运行一次，它就会留在内存中。当你想要操纵稍大的数据集时，这一特性极其方便。使用 FSI，可以在最初加载数据一次，然后持续编码，不需要像在 C#中那样，每次更改之后都重新加载。

你可能会注意到 x+100 以后神秘的“;;”。这指示 FSI 到目前为止输入的命令都必须执行。如果你想要执行的代码跨越多行，这就很有用。

■ **提示**：如果你尝试过在 FSI 中直接输入 F#代码，可能会注意到其中没有智能感知功能。与完整的 Visual Studio 体验相比，FSI 是较为粗糙的开发环境。我的建议是尽量少在 FSI 中输入代码，而主要在.fsx 文件中工作，这样就能获得现代 IDE 的所有好处，例如自动完成和语法验证。这将自然地引导你编写完整的脚本，在未来重新执行。虽然脚本不是解决方案构建的一部分，但是它们是解决方案的一部分，也可以（应该）进行版本控制，这样，你就能够重复脚本中进行的任何试验。

1.5.2　创建第一个 F#脚本

我们已经了解了 FSI 的基础知识，下面就可以开始编码了。我们将从读取数据开始，转换 C#示例。首先，我们将执行一个完整的 F#代码块以观察它的操作，然后详细检查所有部分是否都正常工作。删除当前 Script.fsx 中的所有内容，并输入程序清单 1-10 中的 F#代码。

程序清单 1-10　从文件中读取数据

```
open System.IO

type Observation = { Label:string; Pixels: int[] }

let toObservation (csvData:string) =
    let columns = csvData.Split(',')
    let label = columns.[0]
    let pixels = columns.[1..] |> Array.map int
    { Label = label; Pixels = pixels }

let reader path =
    let data = File.ReadAllLines path
    data.[1..]
    |> Array.map toObservation

let trainingPath = @"PATH-ON-YOUR-MACHINE\trainingsample.csv"
let trainingData = reader trainingPath
```

这几行 F#语句执行相当多的操作。在讨论它们的工作原理之前，我们先运行这些语句，观察结果。选中代码，单击鼠标右键并选择“交互执行”，几秒之后，应该看见 F# Interactive 窗口中显示如下内容：

```
>
type Observation =
  {Label: string;
   Pixels: int [];}
val observationFactory : csvData:string -> Observation
```

```
val reader : path:string -> Observation []
val trainingPath : string =
  "-"+[58 chars]
val trainingData : Observation [] =
  [|{Label = "1";
     Pixels =
      [|0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0;
        0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0;
        0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0;
        0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0;
        0; 0; 0; 0; ...|];};
    /// Output has been cut out for brevity here ///
    {Label = "3";
     Pixels =
      [|0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0;
        0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0;
        0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; ...|];};
  ...|]
>
```

本质上，在这十几行 F#代码中，我们得到了 DataReader 和 Observation 类的所有功能。在 F# Interactive 中运行代码，就可以立即加载数据并看到它们的样子。此时，我们在 F# Interactive 会话中加载了一个 Observation（数据）数组，它将根据你的需要停留任意长的时间。例如，假定你想要知道训练集中第 100 号观测值的标签，没有必要重新加载或者重新编译，只需要在 F# Interactive 窗口中输入如下语句并执行：

```
let test = trainingData.[100].Label;;
```

这样就行了，因为数据已经在内存中，随时可以使用。

交互执行极其方便，尤其是在数据集很大、加载很费时的情况下。这是 F#在以数据为中心的工作中相对于 C#的明显好处之一：C#控制台应用程序中的代码有任何更改，都需要重新构建和重新加载数据，而 F# Interactive 加载数据之后，你就可以使用这些数据，更改核心内容了。更改代码和试验时无须重新加载。

1.5.3 剖析第一个 F#脚本

现在，我们已经看到了 10 行代码得到的结果，下面深入它们的工作原理：

```
open System.IO
```

这一行很简单——等价于 C#中使用 System.IO 的语句。F#可以访问每个.NET 库，所以多年学习.NET 命名空间积累的经验都不会白费——你可以重用所有库，并用 F#特有的优良功能扩增它们！

在 C#中，我们创建一个 Observation 类以保存数据。在 F#中，使用稍有不同的类型完成相同的工作：

```
type Observation = { Label:string; Pixels: int[] }
```

砰！完成了！在这一行代码中，我们创建一个记录（F#特有的类型）。记录本质上是一个不可变的类（如果从 C#调用 F#代码，它看上去就是这样的），有两个属性：Lable 和 Pixels。记录的使用很简单：

```
let myObs = { Label = "3"; Pixels = [| 1; 2; 3; 4; 5 |] }
```

我们简单地使用花括号，填写所有属性，实例化 Observation 记录。

F#自动推导出我们想在 Observation 中放入的数据，因为它是具有正确属性的唯一记录类型。我们简单地用[| |]表示数组的开始和结尾，并填写内容，创建用于 Pixels 属性的整数数组。

现在，我们已经有了数据容器，可以从 CSV 文件中读取数据了。在 C#示例中，我们创建一个方法 ReadObservations 以及包含该方法的 DataReader 类，但是老实说，该类对我们来说作用不大。所以我们将简单地编写一个 reader 函数，以路径作为参数，并使用一个辅助函数从 CSV 行中提取观测值：

```
let toObservation (csvData:string) =
    let columns = csvData.Split(',')
    let label = columns.[0]
    let pixels = columns.[1..] |> Array.map int
    { Label = label; Pixels = pixels }

let reader path =
    let data = File.ReadAllLines path
    data.[1..]
    |> Array.map toFactory
```

我们在这里使用了相当多的 F#特性——下面将一一解说。这些功能的出现有些密集，但是一旦你仔细理解，就已经学到用 F#高效地进行数据科学研究所需理解知识的 80%了！

我们从大致的概况开始。下面是等价的 C#代码（程序清单 1-2）。

```
private static Observation ObservationFactory(string data)
{
    var commaSeparated = data.Split(',');
    var label = commaSeparated[0];
    var pixels =
        commaSeparated
        .Skip(1)
        .Select(x => Convert.ToInt32(x))
        .ToArray();

    return new Observation(label, pixels);
```

```
}

public static Observation[] ReadObservations(string dataPath)
{
    var data =
        File.ReadAllLines(dataPath)
        .Skip(1)
        .Select(ObservationFactory)
        .ToArray();

    return data;
}
```

C#和 F#之间有一些明显的差异。首先，F#不使用花括号；和其他语言（如 Python）一样，F#使用空格表示代码块。换言之，F#代码中的空格很重要：当你看到空格缩进深度相同的代码时，它们就属于同一个代码块，仿佛它们之间围绕着不可见的花括号一样。在程序清单 1-10 中的 reader 函数中，我们可以看到函数体从 let data 开始，以|>Array.map observationFactory 结束。

另一个明显的不同之处是函数参数中没有返回类型或者类型声明。这是不是意味着，F#是一种动态语言？如果将鼠标指针放在.fsx 文件中的 reader 之上，就会显示如下提示：val reader: path:string ->Observation []。这表示一个函数取字符串类型（string）参数 path，返回一个 Observation 数组。和 C#一样，F#是静态类型语言，但是使用一个强大的类型推理引擎，该引擎利用任何可用的提示，自行理解正确的类型。在本例中，File.ReadAllLines 只有两次重载，唯一可能的匹配暗示 path 是一个字符串。

在某种程度上，这使你两者兼得——你得到了动态语言代码较少的好处，而且还有一个稳定的类型系统，由编译器帮助你避免愚蠢的错误。

■ **提示**：F#的类型推理系统利用少数提示理解意图的能力绝对令人惊叹。但是，有时候它无法自行推导出结果，必须为它提供帮助。在那种情况下，你可以用预期类型加以注释，如：let reader (path:string)=。一般来说，我建议，即使没有必要，也在代码的高层组件或者关键组件中使用类型注释。这样有助于其他人更直接地理解你的代码意图，而不需要打开 IDE 查看推导的类型。类型注释还有助于确认在组合多个函数时，每一步都真正地传递预期的类型，以跟踪某些问题的来源。

C#和 F#的另一个有趣的差异是没有返回语句。和大体上是过程性语言的 C#不同，F#是面向表达式的。let x=2+3*5 这样的表达式将名称 *x* 与一个表达式绑定；当表达式求值（2+3*5=17）时，值与 *x* 绑定。函数也是如此：函数的值就是最后一个表达式的值。下面是说明此时发生情况的人为例子：

```
let demo x y =
    let a = 2 * x
    let b = 3 * y
    let z = a + b
```

```
    z // this is the last expression:
      // therefore demo will evaluate to whatever z evaluates to.
```

你可能已经发现了另一个差异——参数周围也没有括号。 我们暂时忽略这一点，本章后面再进行说明。

1.5.4　创建函数管道

下面我们深入观察读取函数的主体。let data = File.ReadAllLines path 一次性将位于 path 所指路径的文件的所有内容读入一个字符串数组，每行为一个字符串。这里没有什么魔法，只是证明我们确实可以在 F#中使用.NET 框架中的任何可用功能，而不用理会编写那些功能的语言。

data.[1..]说明了 F#中的索引语法，myArray.[0]将返回数组的第一个元素。注意，myArray 和带括号的索引之间有一个点！这里要做说明的另一个有趣的语法特征是数组切片。data.[1..] 表示“给我一个新数组，从索引 1 处开始获取数据，直到最后一个元素。”类似地，可以采用 data.[5..10]（给我从索引 5 到索引 10 的所有元素）或者 data.[..3]（给我到索引 3 为止的所有元素）。这使得数据操纵极其方便，是 F#成为很好的数据科学语言的原因之一。

在我们的例子中，所有元素都从索引 1 开始，换言之，我们丢弃数组的第一个元素——表头。

C#代码中的下一步用 ObservationFactory 方法从每一行中提取一个 Observation 值，使用的语句如下：

```
myData.Select(line => ObservationFactory(line));
```

F#中的等价语句是：

```
myData |> Array.map (fun line -> toObservation line)
```

你可以粗略地将这条语句理解为“将 myData 数组传递给一个函数，该函数对每个数组元素应用映射，将每行转换为一个观测值。”这与 C#代码有两处重要的不同。首先，虽然 Select 语句作为所操纵数组上的一个方法出现，但是在 F#中，函数的逻辑所有者不是数组本身，而是 Array 模块。Array 模块的工作方式与提供 IEnumerable 上的一组 C#扩展方法的 Enumerable 类相似。

注意：如果你喜欢在 C#中使用 LINQ，我怀疑你会真的喜欢 F#。在许多方面，LINQ 将来自函数式编程的概念导入面向对象的 C#语言。F#为你提供更深入的类 LINQ 功能。要对此有所感觉，只需要在.fsx 中输入“Array”，看看可以使用多少种函数！

第二个重要的差异是神秘的“|>”符号。该符号称为“**管道向前**”操作符。简而言之，它将前一个表达式的结果传递给管道中的下一个函数，后者使用它作为最后一个参数。例如，考虑如下代码：

```
let double x = 2 * x
let a = 5
let b = double a
let c = double b
```

上述代码可以重写为：

```
let double x = 2 * x
let c = 5 |> double |> double
```

double 函数只需要一个整数型参数，因此我们可以通过管道向前操作符直接“馈送”数值 5。因为 double 的结果也是整数，所以我们可以继续向前传递结果。可以将这段代码重写为：

```
let double x = 2 * x
let c =
    5
    |> double
    |> double
```

Array.map 示例遵循相同的模式，如果我们使用原始的 Array.map 版本，代码为：

```
let transformed = Array.map (fun line -> toObservation line) data
```

Array.map 需要两个参数：应用到每个数组元素的转换，以及应用转换的数组。因为目标数组是函数的最后一个参数，所以我们可以使用管道向前功能“馈送”要映射的数组，如：

```
data |> Array.map (fun line -> toObservation line)
```

如果你对 F#完全不熟悉，可能有些不知所措。不要担心！我们将逐步看到许多 F#的示例，理解其中所使用语句的原理可能要花点时间，但是入门实际上相当容易。管道向前操作符是我最喜欢的 F#功能之一，因为它使工作流变得很容易跟踪：取得某些数据，沿着各步骤或者运算组成的管道传递，直到工作完成。

1.5.5 用元组和模式匹配操纵数据

我们已经有了数据，现在必须找出训练集中最接近的图像。正如 C#示例中那样，我们需要图像的距离。和 C#不同，我们不创建类或接口，而只使用函数：

```
let manhattanDistance (pixels1,pixels2) =
    Array.zip pixels1 pixels2
    |> Array.map (fun (x,y) -> abs (x-y))
    |> Array.sum
```

这里我们使用了 F#的另一个核心功能：元组和模式匹配的组合。**元组**（**Tuple**）是一组无名称的有序值，类型可能不同。元组在 C#中也存在，但是该语言中缺乏模式匹配，确实削弱了元组的实用性，这很令人遗憾，因为这两种功能是数据操纵的黄金组合。

同时使用模式匹配比起只使用元组要强大得多。一般来说，模式识别是一种机制，使你的代码能够简单地识别数据中的各种形状，并据此采取行动。下面是一个说明模式识别在元组上作用方式的小例子：

```
let x = "Hello", 42 // create a tuple with 2 elements
let (a, b) = x // unpack the two elements of x by pattern matching
printfn "%s, %i" a b
printfn "%s, %i" (fst x) (snd x)
```

这里我们在 *x* 中“打包”了两个元素：字符串“Hello”和整数 42，用逗号分隔。逗号通常表示一个元组，因此在 F#代码中必须注意这一点，初看时会稍有些混乱。

第二行“解包”元组，将两个元素读入 *a* 和 *b*。对于两个元素的元组，存在一种特殊的语法：可以使用 fst 和 snd 函数访问第一个和第二个元素。

提示：你可能已经注意到，和 C#不同，F#不使用括号定义函数参数列表。举个例子，加法函数通常这样编写：add x y= x+y。函数 tupleAdd (x, y) = x + y 是完全有效的 F#代码，但是含义不同：它需要一个参数，该参数是完整形式的元组。因此，1 |> add 2 是有效的代码，但是 1 |> tupleAdd 2 无法编译——而(1,2) |> tupleAdd 可以正常工作。

这种方法可以扩展到超过 2 个元素的元组，主要的不同之处是不支持 fst 和 snd。注意 wildcard _ below 的使用，它的意思是“忽略第 2 个位置的元素”：

```
let y = 1,2,3,4
let (c,_,d,e) = y
printfn "%i, %i, %i" c d e
```

我们来看看这在 manhattanDistance 函数中的效果。我们取得两个像素（图像）数组并应用 Array.zip，创建一个元组数组，相同索引的元素被配对在一起。简单地举个例子可能有助于理解：

```
let array1 = [| "A";"B";"C" |]
let array2 = [| 1 .. 3 |]
let zipped = Array.zip array1 array2
```

在 FSI 中运行上述代码，应该产生如下输出，无须另加说明：

```
val zipped : (string * int) [] = [|("A", 1); ("B", 2); ("C", 3)|]
```

因此，曼哈顿距离函数所完成的就是取两个数组，配对对应的像素，对每一对像素计算差值的绝对值，然后加总所有值。

1.5.6　训练和评估分类器函数

现在我们有了曼哈顿距离函数，就可以搜索训练集中最接近所分类图像的元素了。

```
let train (trainingset:Observation[]) =
    let classify (pixels:int[]) =
        trainingset
        |> Array.minBy (fun x -> manhattanDistance x.Pixels pixels)
        |> fun x -> x.Label
    classify

let classifier = train training
```

train 函数以一个 Observation 数组为参数。在函数中，我们创建另一个函数 classify，以一个图像为参数，寻找与目标距离最小的图像，并返回最接近的候选图像的标签；train 返回该函数。还要注意，manhattanDistance 虽然不是 train 函数的一部分，但是仍然可以在该函数中使用，这称作“将变量捕捉到一个闭包中”，在一个函数中使用该函数中未定义作用域的变量。minBy（不存在于 C#或者 LINQ 中）的用法也需要注意，该函数使用很方便，我们可以将它用于任何比较数值项的函数，找出数组中最小的元素。

现在，简单地调用 train training 就可以创建一个模型。

将鼠标指针悬停在 classifier 上，就可以看到它的类型：

```
val classifier : (int [] -> string)
```

上述显示告诉你，classifier 是一个函数，以一个整数数组（你打算分类的图像像素）为参数，返回一个字符串（预测的标签）。一般来说，我高度建议花一些时间将鼠标指针放在代码上，确保类型和你想象的一样。F#类型推理系统极其出色，但是有时候它过于聪明了，能够想出让你的代码正常工作的方法，但是可能不是你所预计的方法。

我们已经完成大部分工作了，现在验证分类器：

```
let validationPath = @" PATH-ON-YOUR-MACHINE\validationsample.csv"
let validationData = reader validationPath

validationData
|> Array.averageBy (fun x -> if model x.Pixels = x.Label then 1. else 0.)
|> printfn "Correct: %.3f"
```

这就很简单地转换了我们的 C#代码：读取验证数据，用 1 标记每个正确的预测，并计算平均正确率，完成了！

在一个文件的 30 行代码中，我们就得到了所有必要的功能。

F#的能力明显比我们在这堂速成课中看到的强得多。但是，现在你应该已经对 F#语言的概念及其适合于机器学习和数据科学的原理有了更好的认识。F#代码简短但是容易理解，数据转换管道工作得很出色，它是机器学习中必不可少的活动。F#Interactive 窗口可以在内存中加载数据一次，然后研究数据、建立思路模型，而无须浪费时间重新加载和编译。仅仅这些就是巨大的好处了——但是我们将在本书后面的章节中看到更多 F#的相关知识，以及结合它与 C#能力的方法！

1.6　改进我们的模型

我们实现了可能有效的最愚蠢的模型，它也确实干得不错——93.4%的正确率。我们还能让它变得更好吗？

遗憾的是，对此没有通用的答案。除非你的模型预测已经 100%正确，否则始终有可能改进，知道答案的方法只有一个：尝试各种方法，看看它们是否有效。构建好的预测模型涉及许多反复尝试，正确建立模型以快速迭代、试验和验证思路至关重要。

我们可以探索哪些方向？我可以不假思索地提出一些。

- 调整距离函数。我们在此使用的曼哈顿距离只是许多可能性中的一个，选择“正确”的距离函数通常是获得好模型的关键因素之一。距离函数（或者代价函数）本质上是向机器传达如何在其世界中考虑类似或者不同事物的手段，因此认真思考这一点是非常重要的。
- 搜索多个最靠近的点，而不是只考虑一个最近点，并进行“多数表决”。这可使模型变得更加健壮；观察更多的候选可以降低无意中选择错误对象的概率。这种方法有一个名称——“K 最近邻”算法，是机器学习的经典方法之一。
- 对图像进行一些巧妙的处理。例如，想象拍摄一张照片，但是向右移动一个像素。如果将这个图像与原始版本比较，尽管它们是同一个图像，但距离可能很大。用于补偿这一问题的方法之一是使用某种模糊。用邻近像素颜色的平均值代替每个像素能够缓解“图像不重合”问题。

我敢肯定，你也能想到其他的主意，下面我们一起来探索第一种思路。

1.6.1　试验距离的另一种定义

我们从距离开始。为何不尝试一下高中已经学过的距离（学术上叫作“欧几里得距离”）？下面是这种距离的公式：

$$Dist(X,Y)=\sqrt{(x_1-y_1)^2+(x_2-y_2)^2+\cdots+(x_n-y_n)^2}$$

上述公式说明，两点 X 和 Y 之间的距离是对应坐标差值平方和的平方根。你可能已经见过这一公式的简化版本，如果取某个平面上的两个点：$X=(x_1, x_2)$ 和 $Y=(y_1, y_2)$，其欧几里得距离为：

$$Dist(X,Y)=\sqrt{(x_1-y_1)^2+(x_2-y_2)^2}$$

相比数学，代码可能让你更舒服一些，下面是上式在 F#中的表现形式：

```
let euclideanDistance (X,Y) =
```

```
    Array.zip X Y
    |> Array.map (fun (x,y) -> pown (x-y) 2)
    |> Array.sum
    |> sqrt
```

我们取两个浮点数组作为输入（每个数组代表一个点的坐标），计算每个元素差值的平方，加总，再求平方根。这不是很难，而且相当清晰！

在此应该提到几个技术细节。首先，F#内建了许多很好的数学函数，通常可以在System.Math类中搜索。Sqrt就是这样一个函数——用let x = sqrt 16.0代替var x = Math.Sqrt(16)不是好多了吗？pown是另一个此类函数，它是指数为整数时“求 *n* 次方”的专用版本，通用版本则是**运算符，如let x = 2.0 ** 4.0。如果已知指数为整数，pown将明显提升性能。

另一个细节，我们在此使用的距离函数是正确的，但是从技术上说，对于我们的目的，实际上可以去掉sqrt。我们需要的是最接近的点，如果0≤A<B，则sqrt A<sqrt B。因此不必承担这一运算的代价，可以将其去掉。这使我们仅在整数上运算，比双精度数或者浮点数要快得多。

1.6.2 重构距离函数

如果我们的目标是试验不同的模型，现在就是进行一些重构的好时机。我们希望更换代码中的不同部分，观察对预测质量的影响。具体地说，我们打算切换距离。典型的面向对象方法是提取一个接口（如IDistance），注入训练中（这就是C#示例中所做的）。然而，如果你真正地思考了这个问题，就会发现接口有些大材小用了——我们需要的只是一个函数，以两个点作为输入并返回一个整数——两点之间的距离。可以这样做。

程序清单 1-11 重构距离函数

```
type Distance = int[] * int[] -> int

let manhattanDistance (pixels1,pixels2) =
    Array.zip pixels1 pixels2
    |> Array.map (fun (x,y) -> abs (x-y))
    |> Array.sum

let euclideanDistance (pixels1,pixels2) =
    Array.zip pixels1 pixels2
    |> Array.map (fun (x,y) -> pown (x-y) 2)
    |> Array.sum
let train (trainingset:Observation[]) (dist:Distance) =
    let classify (pixels:int[]) =
        trainingset
        |> Array.minBy (fun x -> dist (x.Pixels, pixels))
        |> fun x -> x.Label
    classify
```

我们创建一个 Distance 类型以代替接口，这是一个函数签名，以一对像素为参数，返回一个整数。现在可以传递任意多个 Distance 作为 train 函数的参数，这将返回使用特定距离的分类器。例如，我们可以用 manhattanDistance 函数或者 euclideanDistance 函数（或者想要进一步试验的任意距离函数）训练分类器，比较它们的表现。注意，这在 C#中也完全可能做到。我们可以简单地使用 Func<int[],int[],int>，而不创建 IDistance 接口，下面是代码。

程序清单 1-12　函数式 C#示例

```
public class FunctionalExample
{
    private IEnumerable<Observation> data;

    private readonly Func<int[], int[], int> distance;

    public FunctionalExample(Func<int[], int[], int> distance)
    {
        this.distance = distance;
    }

    public Func<int[], int[], int> Distance
    {
        get { return this.distance; }
    }

    public void Train(IEnumerable<Observation> trainingSet)
    {
        this.data = trainingSet;
    }

    public string Predict(int[] pixels)
    {
        Observation currentBest = null;
        var shortest = Double.MaxValue;

        foreach (Observation obs in this.data)
        {
            var dist = this.Distance(obs.Pixels, pixels);
            if (dist < shortest)

            {
                shortest = dist;
                currentBest = obs;
            }
        }
```

```
            return currentBest.Label;
        }
    }
```

这种方法的重点在于组成函数而不是对象，是典型的函数式编程。C#虽然来源于面向对象语言，但是也支持许多函数式方法。可以说，从 3.5 版本开始，C#中的每个特性和创新（LINQ、Async 等）都是从函数式编程中借鉴的。现在，人们在描述 C#时会说，它首先是面向对象的，但是支持函数式方法，而 F#首先是一种函数式编程语言，但是也适应面向对象风格。在本书中，我们一般更强调函数式风格，而不强调面向对象风格，因为按照我们的看法，前者比后者更适合机器学习算法，还有一个原因是，以函数式风格编写的代码通常提供很大的优势。

在任何情况下，我们都要做好设置，观察自己的想法是否表现良好。现在，可以创建两个模型，比较它们的表现：

```
let manhattanClassifier = train trainingData manhattanDistance
let euclideanClassifier = train trainingData euclideanDistance

printfn "Manhattan"
evaluate validationData manhattanClassifier
printfn "Euclidean"
evaluate validationData euclideanClassifier
```

在此，最好的一件事是我们不需要在内存中重新加载数据集，只需要简单地修改脚本文件，选择包含所要运行新代码的部分，运行该部分即可。如果我们那么做，就应该看到新模型 euclideanModel 的分类图像的正确率为 94.4%，而不是 manhattanModel 的 93.4%。好极了！我们取得了 1%的改进。1%看起来似乎不多，但是如果准确率已经达到 93.4%，这就意味着错误率从 6.6%降低到 5.6%，有大约 15%的收获。

从距离函数中的一个小变化，我们得到了很明显的改进。这似乎是一个有前景的方向，可能值得试验更复杂的距离函数，或者尝试前面提到的其他方向。如果不多加小心，就有可能因为多次试验而留下一大堆乱七八糟的代码。版本控制和自动化对经典软件开发是必不可少的，同样，在开发预测模型时，它们也是你的朋友。过程的细节和节奏可能不同，但是总体思路相同：你希望在对代码进行更改时不需要担心造成任何破坏，“在我的机器上可以正常工作”还不够好。在任何时候，任何人都应该可以使用你的代码，在没有任何人工干预的情况下复制你的结果。这种方法称作“可复制研究”。我强烈鼓励你朝着这个方向发展，尽可能在任何地方使用脚本和源代码控制。我自己曾经经历过这样的恐怖场景：前一天你的模型工作得很好，但是因为没有花时间清晰地进行保存，忘记了自己究竟是怎么做到的，从而弄丢了模型。不要成为那样的人！特别是在用 Git 或者 Mercurial 等 DVCS 创建分支如此容易的时代，没有任何借口犯那样的错误。对某个模型有新想法时，可以创建一个分支，保存在脚本中执行的步骤。

1.7　我们学到了什么

我们在本章中介绍了许多基础知识！在机器学习方面，你现在已经熟悉了一些关键概念和方法学。如果你是 F#的新手，现在也已经编写了第一段 F#代码！

下面回顾一些要点，先从机器学习方面开始。

首先，我们讨论了交叉验证——使用不同数据集进行训练和验证，留出某些数据评估预测模型质量的过程。这从许多方面看都是关键的过程。首先，它为你提供一个基准——指导试验的“真实状态”。没有验证集，就无法判断特定模型是否比另一个模型好。交叉验证可以科学地计量质量，它的作用类似于自动化测试套件，可以在开发工作偏离方向时提出警告。

建立了交叉验证机制，就可以在可量化的基础上试验、选择模型或者方向。反复尝试方法是“进行机器学习”的关键部分。没有一种方法能够事先知道特定的方法是否有效，所以必须在数据上尝试，自行观察，因此为成功做好准备，接受“可复制研究”的思路非常重要。尽可能用脚本使你的研究过程自动化，大量使用源代码控制，以便在任何时点、没有人工干预的情况下复制模型的每一步。总而言之，做好准备，使自己能够轻松地更改和运行代码。

1.7.1　在好的距离函数中能找到什么

在数字识别器的探索中，我们发现所使用的距离中的小小变化就能显著地改进模型，大部分（所有？）机器学习模型的核心中都有一个距离函数。所有学习过程归根结底都是计算机找出为特定问题搭配已知数据的最佳方式的尝试——“最佳搭配”的定义完全封装在距离函数中。

在某种程度上，距离函数的作用是利用数学方法，将你的目标从人类理解的形式转换成机器理解的语句。距离函数也常常被称作代价函数，从代价上考虑更多地强调了“坏”解决方案之所以不好的原因——所招致的惩罚，帮助我们避免选择不好的解决方案。

在我的经验中，花时间思考代价函数总是值得的。不管使用的算法有多巧妙，如果代价函数有缺陷，结果就很糟糕。想象一个包含两个计量值的个人数据集：以英尺为单位的身高和以磅为单位的体重。如果我们按照图像识别中使用的方法，在数据集中搜索“最相似的人”，就会发生这样的情况：身高的范围通常在 5～6 英尺（1 英尺 0.3048 米）之间，而体重的范围更广，差别更大，例如 100～200 磅（1 磅=0.4536 千克）。因此，直接根据两个计量值之间的差值计算距离实际上将忽略身高的差别，因为 1 英尺的差别等价于 1 磅的差别。解决这个问题的方法之一是转换所有身体特征，确保它们处于一致的比例——这一过程称作“规格化”，是后面将要详细讨论的一个主题。幸运的是，所有像素都在相同的标度上编码，在本章中可以忽略这个问题，但是我希望这个“距离产生错误”的例子能够让你认识到，为什么距离函数需要深思熟虑！

这也是正确的数学定义真正起作用的情况之一。如果你翻起在学校时的数学笔记，就会看到距离函数（数学家们有时候称之为“度量标准”）由几个属性定义：

- Distance(A,B) > 0（距离不能为负数）。

- 当且仅当 *A*=*B* 时，Distance(A,B) = 0（只有 *A* 和自己的距离为 0）。
- Distance(A,B) = Distance(B,A)（对称性：从 *A* 到 *B* 的距离等于从 *B* 到 *A* 的距离）。
- Distance(A,B) <= Distance(A,C) +Distance(C,B)（“三角不等式”：两点之间直线最短）。

在本章中我们只关注两种距离，但是满足上述属性的函数多种多样，每一种都以不同的方式定义了相似性。因此，模型中的代价函数不需要满足所有属性，但是一般来说，如果不满足某些属性，你应该问问自己可能会因此引起什么意外的副作用。例如，在曼哈顿距离的第一个例子中，如果我们忽略了绝对值，就明显地违反了规则 1（距离非负）和规则 3（对称性）。在某些情况下，随意使用不为度量指标的函数有充足的理由，但是发生这种情况时，应该多花一点时间思考可能出现问题的地方！

1.7.2 模型不一定要很复杂

最后，我希望强调的是，有效的模型并不一定很复杂！不管在 C#还是 F#中，两个分类器都很小，使用的是相当简单的数学方法。当然，有些复杂模型也能提供惊艳的结果，但是如果能用容易理解和修改的简单模型得到相同的结果，那么为什么不省点事呢？

这一原则称作“奥卡姆剃刀”，名称来源于中世纪哲学家“奥卡姆的威廉”。奥卡姆剃刀遵循经济原理。试图解释某一事物时，在多个可能合适的模型中选择最简单的一个。只有在简单的解释无法奏效时，才选择复杂的模型。

同理，我们首先实施“可能有效的最简方法”，我很鼓励你遵循这一方法。如果不这样做，可能会发生这样的情况：你开始实施几个可能有效的方法，它们激发出新的思路，因此很快你将有许多不成熟的原型，在丛林中越陷越深，而没有清晰的过程或者方法。突然之间，你将意识到自己已经为编码花费了几周，却还没有确定任何有效的方法或者前进的方向。这不是一种好的感觉。管理自己的时间，花费一天（或者一周、一小时——任何对你的问题来说切合实际的时间）用能够想到的最蠢、最简单的预测模型建立一个端到端模型。这个模型可能已经足够好了，这样的情况下就不会浪费任何时间。如果它不够好，此时你已经有了合适的机制，准备了数据集成和交叉验证，也很有可能已经发现了数据集中存在的意外问题。你将从合适的位置进入丛林。

1.7.3 为什么使用 F#?

如果这是你第一次遭遇 F#，开始时可能会觉得奇怪：我为什么介绍那种语言，而不坚持使用 C#? 我希望书中的例子清楚地说明了原因！在我看来，C#虽然是一种很出色的语言，但是 F#和机器学习及数据探索的搭配简直不可思议！在接下来的几章中将看到 F#的更多特性，但是我希望这些理由在本章中就已经很清晰。

首先，F#的交互执行和脚本环境绝对可以节约时间。在开发机器学习模型时，快速试验、更改代码、查看效果是至关重要的，因此需要一个脚本环境。我的典型 F#工作流是在每天开始时将数据加载到交互执行环境中。我不需要再次加载它们——数据已经在内存中，可以自

由地摆弄模型。相比之下，在 C#中测试新思路迫使我重建代码并重新加载数据，这最终会成为一个巨大的时间陷阱。

而且，F#的一些语言特性使其极适合于数据操纵。它聚焦于函数而非对象，擅长将函数应用到数据，以任何需要的方式组成函数。管道向前操作符|>对组合这些转换很方便，创建了一个管道，以非常容易理解的方式表现数据转换工作流。结合将数据打包为元组和用模式匹配解包元组的能力，以及 Array、List 和 Seq 模块中大量的内建函数，就可以得到和 LINQ 一样的增强功能，以任何方式转换和改造数据。

1.8　更进一步

人们第一次看到算法时，我常常看到的反应之一是“但是……这有点蠢。”他们真正的意思是，“但是……哪里有学习？”机器学习的正式定义是：“编写一个程序，在得到更多数据时，更好地执行某项任务。”我们的模型——1-最近邻算法完全符合这一定义：可以预期，越大的样本“工作得越好”（反过来，可以预期减少样本会使性能降级）。另一方面，它不太符合我们对“学习”的直观理解。我们的模型实际上记住了看到的所有情况，而人们倾向于将学习视为对高层抽象的辨别，这提供了经验的更紧凑表现形式。当你看到一个数字的图像时，不会重温一生中见过的所有图像以决定是否匹配——你知道用于过滤和识别的高层概念（“0 就像一个圆圈”）。

可以在我们的问题上使用更高级的算法（如支持向量机或者神经网络，第 8 章中将这么做），它们的行为和“学习”的概念更符合：在训练阶段处理数据，并提取简化的表现形式。这样做的明显缺点是训练阶段实际上将变得更加复杂，好处是所得到的模型更小、更快。

那么，应该使用支持向量机、神经网络还是其他方法？和往常一样，答案是“看情况”。首先，这取决于你的终极目标。你的目标可能是更加精确；如果代码部署于生产环境，目标也可能是更快、使用更少内存或者其他。

正如前面所讨论的那样，实际上没有办法知道某种模型是否好于另一种——你必须尝试，这可能代价很高。所以，在追求更好的模型之前，要仔细考虑：当前模型是否足够好？工作是否集中于正确的问题？正如法国谚语所说“更好是好的敌人”。我们的简单算法已经达到大约 95%的分类正确率，相当不错了。如果这已经足够，就不要浪费时间，转向感兴趣的下一个问题。机器学习问题有能力自己生存，除非你有一个快速、100%准确的完美模型，否则永远不能说“完成”了，回到工作中继续改善、挤出每一点精确度是很有诱惑的。因此，必须提前想好什么是“足够好”，否则可能需要很长时间来追求完美！

实用链接

- F#软件基金会（www.fsharp.org）提供关于 F#设置与入门的丰富信息。
- Kaggle 数字识别器竞赛的完整数据集可以在 www.kaggle.com/c/digit-recognizer 中找到。你甚至可以进入比赛，看看会遇到什么——这很有乐趣，也是很棒的学习经历！

第 2 章

垃圾邮件还是非垃圾邮件？

用贝叶斯定理自动检查垃圾邮件

如果你使用电子邮件（我想你一定用），有可能每天都在工作中看到机器学习。你的电子邮件客户端可能包含某种垃圾邮件过滤机制，从收到的邮件中自动识别令人厌烦的推销材料，然后小心地将它们发送到“垃圾邮件文件夹”。这个功能免去了人工逐个删除这些邮件的麻烦，还可以防止你不小心点击有害链接，是机器学习实施得当令生活更美好的典型范例。计算机擅长执行重复性任务，并且做得很彻底。通过自动进行乏味的活动，避免我们犯错，使人们可以专注于更有趣、更值得深思的活动。

垃圾邮件检测是机器学习的极好例证，也是分类概念的权威示例。我们必须将给定的一组电子邮件分为两类：垃圾邮件（应该直接进入垃圾箱的有害信息）或者非垃圾邮件（我们希望阅读的有效信息）。这一工作还有另一个有趣之处：和第 1 章处理的数字识别问题不同，垃圾邮件检测所用的数据都是文本，而不是数字形式。在数字世界中，大部分可用材料是文本而不是结构清晰的数字表格，因此理解处理方式十分重要。

在本章中，我们将从头开始构建一个垃圾邮件检测引擎，然后用这个例子说明几种有用的技术和思路，我们将：

- 确定处理文本的方法。从计算机的角度看，原始文本文档就是字符的随机组合。我们将看到如何将一个文本块转换为计算机所能处理的标记（Token），从中提取特征（也就是将原始数据转换为信息量更大、更有用的新属性——“特征”）。
- 了解贝叶斯定理——这个简单而强大的数学公式可以量化所包含的新信息，以更新我们对不确定事件的评估。作为直接的应用，我们将实现一个简单贝叶斯分类器——使用词汇频度确定文档类型的算法。
- 讨论理解数据以正确解读结果及其质量的重要性，并阐述即使不修改算法本身，仅从数据集提取新特征就能给模型的预测能力带来多么显著的提高。

我们将首先专注于实现一个分析文本文档，确定其所属类别的算法。一旦有了这个工具，我们将把注意力转向数据集，看看仔细检查所拥有的数据如何帮助我们找出新特征，显著改善预测的质量。

2.1　挑战：构建一个垃圾邮件检测引擎

我们将要处理的文档不是电子邮件，而是文本消息。我们的目标是使用来自英国的真实 SMS（短消息服务）数据集发现垃圾短信，这些消息是我们从加州大学欧文分校机器学习知识库中找到的。

> 附注：UCI 机器学习知识库由加州大学欧文分校的机器学习与智能系统中心于 1987 年启动。你可以在 http://archive.ics.uci.edu/ml/中找到这个知识库，它包含将近 300 个干净、文档齐备的数据集，可以按照大小、特征类型等条件搜索和组织。这是一个很有趣的机器学习资源，包含了许多常常用作算法性能评估基准的“经典”数据集。

2.1.1　了解我们的数据集

在讨论使用哪个模型之前，首先来观察一下数据。数据集可以从 http://1drv.ms/1uzMplL 下载（这就是 UCI 知识库中原始文件的复制品，原始文件可以在 http://archive.ics.uci.edu/ml/datasets/SMS+Spam+Collection 上找到）。它作为单个文本文件保存，名为 SMSSpamCollection（没有文件扩展名），包含 5574 条真实的文本消息。每行是一个 SMS，标记为“ham”（非垃圾短信）或者“spam”（垃圾短信）。下面是前面几行：

```
ham     Go until jurong point, crazy.. Available only in bugis n great world
la e buffet...Cine there got amore wat...
ham     Ok lar... Joking wif u oni...
spam    Free entry in 2 a wkly comp to win FA Cup final tkts 21st May 2005.
Text FA to 87121 to receive entry question(std txt rate)T&C's apply 08452810075over18's
ham     U dun say so early hor... U c already then say...
ham     Nah I don't think he goes to usf, he lives around here though
spam    FreeMsg Hey there darling it's been 3 week's now and no word back!
I'd like some fun you up for it still? Tb ok! XxX std chgs to send, £1.50 to rcv
```

第一眼的印象是，人们在文本消息中使用的语言明显不同于“标准英语”！例如“U c”这样的片段是“You see”的简写，在普通的词典中可能找不到。

考虑到这一点，我的第一步是尝试获得两组文本之间差别的“直觉”。有没有一些词语在垃圾短信或者非垃圾短信中出现得更频繁？这能够指导我们构建一个智能引擎，自动区分非垃圾短信和垃圾短信。我们首先按照类别（“ham”和“spam”）分解数据集，计算各自的词语出现频率。鉴于这一活动的探索特性，似乎很适合使用 F#脚本。

■ **提示**：花点时间研究一下数据！盲目对数据集应用算法可能工作得不错，但是走不了太远。就像开发应用程序时应该花时间学习领域模型那样，对数据的更深入理解能够帮助你构建更智能的预测模型。

2.1.2 使用可区分联合建立标签模型

数据是首要的，我们将采取和第 1 章相同的方式，但是有一些变化。数字识别数据集是由数字组成的（像素编码和标签都是如此），而这里的数据有一个特征（文本消息本身）和一个标签（“ham”或者“spam”）。我们应该如何表现它们？

可能的方法之一是将标签编码为布尔值，如非垃圾短信表示为“真”（true），垃圾短信表示为“假”（false）。这种标签工作得很完美，但是也有一些缺点：它不能自动说明字段的含义。例如，如果展示这样编码的一个数据记录：

```
True   Ok lar... Joking wif u oni... ,
```

你怎么可能猜出这里的“True”代表什么？另一个潜在的问题是，如果我们需要增加更多的类别（如非垃圾短信、垃圾短信和有歧义的信息），布尔值无法扩展。

机器学习的难点在于不能增加额外、不必要的复杂性。因此，我们将用 F#的**可区分联合**（有时我也将它称作 DU）表示标签。如果你之前从未见过可区分联合，可以近似地将其看作 C#枚举类型。可区分联合和枚举一样，都定义一组独特的情况，但是功能强大得多。这种类比对 DU 不公平，但是足以帮助我们起步。

我们先使用可以在 FSI 中运行的例子，简单地说明 DU 的工作方式。定义 DU 和定义其他类型一样简单：

```
type DocType =
    | Ham
    | Spam
```

在 F# Interactive 窗口运行上述语句（在最后添加;;触发求值），应该看到如下结果：

```
type DocType =
    | Ham
    | Spam
```

上述语句定义了一个简单类型 DocType，它只能取两个值：Spam 或者 Ham。我喜欢可区分联合的主要原因之一是它们和模式匹配配合得很好，可编写以非常清晰的风格描述业务领域的代码。例如，在我们的例子中，训练集中的每个示例都是非垃圾短信或者垃圾短信，并包含真实消息的内容。我们可以将其表示为一个元组，每个示例是一个 DocType 和一个字符串，按照这一路线处理消息示例，可以在 F# Interactive 窗口中尝试：

```
let identify (example:DocType*string) =
    let docType,content = example
    match docType with
    | Ham -> printfn "'%s' is ham" content
    | Spam -> printfn "'%s' is spam" content
```

这个例子只是为了示意目的，但是指出了以后将要遵循的模式。在这个例子中，我们创建了一个小函数，取得一个消息示例并打印其内容以及所属类别。我们首先通过元组模式匹配将消息示例分为两部分（DocType 和内容），然后在 DocType 的两种可能情况上使用模式匹配，在单独“分支”中处理两种情况。在 FSI 中输入：

```
identify (Ham,"good message");;
```

你应该看到如下结果：

```
'good message' is ham
```

在能够更好地决定如何处理真实 SMS 内容之前，这就是我们的数据模型。下面从加载数据集入手！

2.1.3　读取数据集

和第 1 章一样，我们将把大部分时间花在探究数据集，从脚本环境中积极构建和改进模型上。打开 Visual Studio 创建一个新的 F#库项目，将其命名为 HamOrSpam。为了方便起见，我们还从 Visual Studio 之外使用文件系统，在解决方案中添加一个名为 Data 的文件夹，将数据文件 SMSSpamCollection 拖入其中（从前面提到的链接中下载，没有文件扩展名）。参见图 2-1。

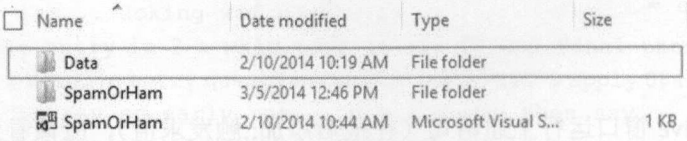


图 2-1　文件夹组织

这就使我们可以使用一个小技巧，用比第 1 章更清晰的方式访问数据文件。F#有两个方便的内建常量__SOURCE_DIRECTORY__和__SOURCE_FILE__，大大简化了脚本中文件的处理，我们可以引用数据文件位置，而无须硬编码一个取决于本地机器的路径。从这些常量的名称你可能已经猜到，第一个常量求得包含文件本身的目录完整路径，第二个返回到文件本身的完整路径，包括文件名。

现在可以进行脚本的编写了。和数字识别器使用的数据集之间主要的差别是，这个数据集没有文件头，只有两列，不使用逗号而使用制表符分隔。其他方面大部分相同：我们有一

个包含示例的文件，希望将其提取到一个数组中。果不其然，解决方案看起来非常相似：读取文件并向每一行应用一个函数，将其解析为标签和内容，根据制表符“\t”拆分。

我们直接进入项目中的 Script.fsx 文件，删除默认的代码并粘贴如下代码：

程序清单 2-1　从文件中读取 SMS 数据集

```
open System.IO

type DocType =
    | Ham
    | Spam

let parseDocType (label:string) =
    match label with
    | "ham" -> Ham
    | "spam" -> Spam
    | _ -> failwith "Unknown label"

let parseLine (line:string) =
    let split = line.Split('\t')
    let label = split.[0] |> parseDocType
    let message = split.[1]
    (label, message)

Let fileName = "SMSSpamCollection"
let path = __SOURCE_DIRECTORY__ + @"..\..\Data\" + fileName

let dataset =
    File.ReadAllLines path
    |> Array.map parseLine
```

此时，你应该可以选择脚本中刚刚编写的所有代码，在 F# Interactive 窗口中执行，产生如下结果：

```
val dataset : (DocType * string) [] =
  [|(Ham,
     "Go until jurong point, crazy.. Available only in bugis n grea"+[50 chars]);
   (Ham, "Ok lar... Joking wif u oni...");
   (Spam,
    "Free entry in 2 a wkly comp to win FA Cup final tkts 21st May"+[94 chars]);
   // Snipped for brevity
   (Ham,
    "Hi. Wk been ok - on hols now! Yes on for a bit of a run. Forg"+[117 chars]);
   (Ham, "I see a cup of coffee animation"); ...|]
```

现在我们有了一个示例数据集，每个都标记为垃圾短信或者非垃圾短信。我们可以开始解决真正感兴趣的问题了：可使用哪些特征区分垃圾短信和非垃圾短信？

提示：使用 File.ReadAllLines 读取数据集的内容可能不总是最好的主意。我们将所有数据加载到内存，马上保存到一个数组中，然后创建另一个数组保存 Array.map 转换的结果。这个特殊数据集相当小，只包含大约 5000 行，但是处理更大的数据集时，可以使用一个流化的版本（如 System.IO.StreamReader.ReadLine()），一次将数据集的一行加载到内存中。

2.2　根据一个单词决定

有了数据，我们就可以开始分析了。我们的最终目标是区分垃圾短信和非垃圾短信，但是和数字识别器的情况不同，我们还没有一组清晰的特征，唯一的材料是原始文本块——SMS 本身。与此同时，人们会猜测文本中有许多信息可供利用。我们只需要找到一个途径，将这些字符串转换成可以使用的特征即可。

2.2.1　以单词作为线索

如果你仔细观察刚刚加载的数据集，就可能会注意到垃圾短信看起来和非垃圾短信有些不同。浏览这些短信，你的眼睛很容易找出某些值得警惕的线索，暗示着某一条短信可能是垃圾短信。举个例子，“FREE”（全部大写）出现在开始的几条垃圾短信中，似乎在非垃圾短信中不常见到。

这就提出了一个可能的方法：使用来自整条短信的单词识别它所属的类别。我们首先计算垃圾短信中包含特定字符串的次数，与非垃圾短信比较，确认我们的直觉是否正确。我们可以在脚本中输入几行 F#代码来测试这一点。首先，过滤数据集使之只包含垃圾短信，然后再次过滤，只保留包含“FREE”的消息，最后计算剩下的条目数量。

```
let spamWithFREE =
    dataset
    |> Array.filter (fun (docType,_) -> docType = Spam)
    |> Array.filter (fun (_,sms) -> sms.Contains("FREE"))
    |> Array.length
```

选择上述代码，在 F# Interactive 窗口中运行——应该看到如下的结果：

```
val spamWithFREE : int = 112
```

现在，让我们对非垃圾短信做同样的工作：

```
let hamWithFREE =
    dataset
    |> Array.filter (fun (docType,_) -> docType = Ham)
    |> Array.filter (fun (_,sms) -> sms.Contains("FREE"))
    |> Array.length
```

这将产生如下结果：

```
val hamWithFREE : int = 1
```

数字确认了我们的直觉：“FREE”在垃圾短信中使用的频率确实远高于非垃圾短信，这似乎是用于区分两个类别的好标志。我们可以使用这个词构建一个很简单的垃圾短信分类器，如：

```
let primitiveClassifier (sms:string) =
    if (sms.Contains "FREE")
    then Spam
    else Ham
```

这是一个很有希望的开始。但是，如果我们想要使用这一思路构建更重要的程序，就需要应付两个问题。首先，我们选择“FREE”作为垃圾短信的指示器，但是并没有真正证明其原因。我们怎么知道一个特定的词汇是不是垃圾或者非垃圾短信的好标志？其次，根据来自整条消息中的一个词做出决定似乎相当有局限性。我们能否一次使用多个词，将它们（可能冲突）的信息组合为单一决策过程？

2.2.2 用一个数字表示我们的确定程度

根据直觉判断包含“FREE”的消息很可能是垃圾短信在统计学上拿不到高学位。我们能不能得到一个不依靠直觉的结论，并且量化特定单词表示一条消息是非垃圾短信或者垃圾短信的可靠性？

我个人常常借助决策树，作为更好地理解概率问题的一种方法。在我们的例子中，下面就是所要做的事情：将语料库中的 5574 个文档按照文档类型分成两组（747 个垃圾文档，4827 个非垃圾文档），对每个组计算包含“FREE”的数量。这可以表示为一棵决策树，如图 2-2 所示。

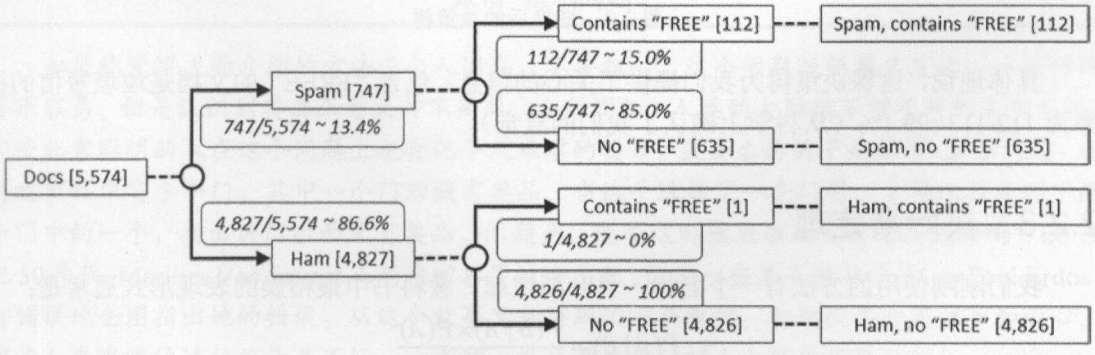


图 2-2 决策树

我喜欢将决策树看成一系列管子（如果你愿意，也可以将其看作管道）。从初始节点开始，在管子中注入 100%的可能性，而不是水。每个节点代表可能有多种结果的一个事件，将各种可能性分解到不同分支，不应该有任何可能性消失。

在本例中，我们从 5574 个文档开始，它们可能是垃圾短信或者非垃圾短信。大部分文档是非垃圾短信——5574 个中有 4827 个，占 86.6%——所以如果没有任何其他信息，随机选择一条消息并问你“是垃圾短信还是非垃圾短信？”你的回答应该是“非垃圾短信”（有 86.6%的概率答对）。

树的下一级考虑每组文档中包含“FREE”的可能性。例如，如果一条消息是垃圾短信，有 112/747 的可能性包含“FREE”，以概率论的术语表达，如果文档是垃圾短信，包含“FREE”的概率是 15%。

这很有趣，但是对我们的分类没有用处。我们真正想知道的是包含“FREE”的文档是垃圾短信的概率。不过，获得这一信息也不太难：我们只需要重新组织决策树，从文档是否包含“FREE”开始，而不是从垃圾短信和非垃圾短信开始。注意，不管如何构造决策树，最终都应该有 4 个“叶子”组。例如，应该仍然有 112 个文档是垃圾短信且包含“FREE”。在 5574 个文档中，有 113 个包含“FREE”，其中 112 个来自垃圾短信组，1 个来自非垃圾短信组。经过更多的重新组织，我们应该得到图 2-3 所示的决策树。虽然和前面的树完全等价，但是提供了不同信息相互关联的不同视角。

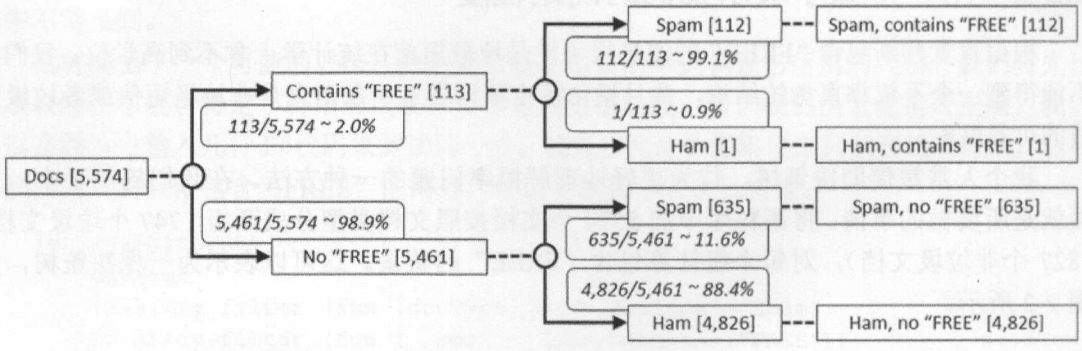


图 2-3　变换后的决策树

具体地说，这棵决策树为我们提供了关心的信息：包含“FREE”的文档是垃圾短信的概率为 112/113=99.1%（从科学上确认了我们的直觉）。

2.2.3　贝叶斯定理

我们刚刚使用的方法有一个名称：贝叶斯定理。教科书中最枯燥的表现形式通常是：

$$P(A \mid B) = \frac{P(B \mid A) \times P(A)}{P(B)}$$

|符号应该读作“在……条件下”。我们刚才的例子计算的是在一封电子邮件包含“FREE”的条件下，该邮件是垃圾邮件的概率，按照贝叶斯公式应该写作：

$$P(\text{SMS是垃圾短信}\mid\text{SMS包含“FREE”})$$
$$=\frac{P(\text{SMS包含“FREE”}\mid\text{SMS是垃圾短信})\times P(\text{SMS是垃圾短信})}{P(\text{SMS包含“FREE”})}$$

如果机械地代入数字，可以得到如下结果，这就是我们用决策树得出的结果：

$$P(\text{SMS 是垃圾短信}\mid\text{SMS 包含“FREE”})=\frac{\frac{112}{747}\times\frac{747}{5574}}{\frac{113}{5574}}=\frac{112}{113}$$

看待贝叶斯定理的方法之一是将其当作两部分数据的平衡措施：对世界的初始认识（在我们的例子中是有多少垃圾短信）以及附加的信息（在垃圾短信或者非垃圾短信中找到单词“FREE”的频率）。贝叶斯定理量化新信息中的权重：垃圾短信越少见，默认的预测就应该越向非垃圾短信倾斜，更换预测方法的证据就越有力。

看待贝叶斯定理的另一种方法是与独立性概念的关联。如果如下等式成立，则称两个事件 A 和 B 相互独立：

$$P(A \text{ and } B)=P(A)\times P(B)$$

使用贝叶斯定理，上式可以得出：

$$P(A\mid B)=P(A)$$

换言之，知道与 B 相关的任何细节不能传达任何关于 A 的有用信息（反之亦然）。相反，如果 A 和 B 不互相独立，贝叶斯定理将捕捉到两部分信息关联的强度。最后一点（也是一句忠告）是，要注意这种效应的不对称性。如果知道一个电子邮件是垃圾邮件，那么它有 15% 的概率包含“FREE”，但是知道一封电子邮件包含单词“FREE”，那么它有 99%的概率是垃圾邮件！

蒙蒂·霍尔问题

如果你觉得上面介绍的方法很令人困惑，不要担心。这个问题既简单又复杂——进行计算很容易，但是认识到其真正意义却不简单。众所周知，人类的大脑很不擅长理解不确定性，即使非常聪明的人在这个问题上也会犯令人难堪的错误。最著名的例子是蒙蒂·霍尔问题：在游戏节目中有 3 个门，其中一个门后藏有奖品，当选手选择了一个门时，主持人打开剩下两个门中的一个，指出该门后面没有奖品。问题是，选手这时候应该改变自己的选择吗？20 世纪 50 年代，Marilyn Vos Savant 正确地回答了这个问题，但是一些著名的数学家（如 Paul Erdos）却错误地企图指出她的错误。从这个故事里我得到了一个教训：处理概率时应该多加小心，因为人类直觉的过往记录并不好——当然，在公开批评任何人之前要三思而行！

2.2.4　处理罕见的单词

贝叶斯定理为我们提供了根据一个单词决定一条短信是非垃圾短信还是垃圾短信的便利方法。但是，你可能已经注意到一个小问题。如果选择仅在一个类别中出现的单词——例如只存在于非垃圾邮件的“uncle”“honeybee”或者“wine”——那么我们的公式将对一个类别指定 100%的概率，另一个类别指定 0%的概率。换言之，如果我们看到一条包含单词“uncle”的消息，我们将带着 100%的自信决定，这条消息不可能是垃圾短信，而不管这条消息的其余内容是什么。

这很明显是不正确的，我们不应该根据有限的训练样本对任何事情持有 100%的信任。处理这一问题的方法之一是使用**拉普拉斯平滑法**。这种听起来很可怕的方法得名于法国数学家皮埃尔-西蒙・德・拉普拉斯（Pierre-Simon de Laplace），可以描述为一个“容差系数”。我们将每个单词的计数加一，消除缺失单词的问题，并计算 P（垃圾短信包含“xyz”）=（1+包含“xyz”的垃圾短信计数）/（1+垃圾短信计数）。本质上，我们制作了一条包含所有标记的虚构消息，因此，最少见的标记出现概率很低，但是不会为 0。

2.3　组合多个单词

使用贝叶斯定理，我们找出了根据是否包含特定单词分类 SMS 的方法。这是一个进展，但是我们还远没有完成任务。一条消息中有许多单词，每个都以不同的强度表现非垃圾短信和垃圾短信的特征。如何将这些信息（它们可能传递相互矛盾的信号）组合为用于决策的单一值?

2.3.1　将文本分解为标记

举个例子，考虑下面这条假设的短信：“Driving, cant txt.”它是垃圾短信的概率有多高?

我们可以直接应用贝叶斯定理，在数据集中搜索整条短信。但是，这种方法估计不会有很好的效果。找到精确复制的概率很小，更不要说多个单词的精确复制了。另一方面，我们已经看到单词“FREE”携带一些相关信息。

我们要做的就是简单地将短信分解为一组单词：“driving”“cant”和“txt”，并用某种表现形式代替原始文本。然后，分析单独标记并将它们所代表的组合为单一值，据此做出决定。

将文本块分解为有意义的元素（单词或者其他符号）——标记（Token）的过程称作**标记化**。决定如何标记化是模型的重要部分，没有任何独特的正确方法——取决于所要解决的问题。现在，我们和以往一样从能够实施的最简单方法入手：如果一个单词“FREE”的信息量很丰富，扩展这一想法，将短信分解为单个单词。实际上，我们将把每条短信转换为一组特征，每个特征中存在或者不存在某个特定单词。

例如，如果我们有两个字符串“Programming is Fun”（编程很有趣）和“Functional programming is more fun!”（函数式编程更有趣！），可以将它们分解为“programming”“is”“fun”和“functional”“programming”“is”“more”“fun”。这种表现方式使我们可以更加高效，不需要检查某条消息是否包含特定的子字符串，可以将每个文档处理为一组标记，快速验证是否包含我们感兴趣的特殊标记。

■ **注意**：这一通用方法对于将文本转换为计算机可以处理的形式很有用。例如，我们可以收集语料库中找到的整组标记——[“functional”; “programming”; “is”; “more”; “fun”]——并通过计算每个标记找到的次数，对每个文档进行编码。第一个文本块变成[0; 1; 1; 0; 1]，第二个文本块变成[1; 1; 1; 1; 1]。我们的数据集现在采用了人类完全不可读的形式，和第 1 章中的数字数据集很相似：每一行是一个观测值，每一列是一个特征，所有内容都编码为数字——例如，我们可以用这些数据确定两个文档的相似度。将观测值转换为一组计算机可以分析寻找模式的数字型特征（人类无法做到），是机器学习中很典型的做法。

这时，你可能会想：“这是严重的过度简化”，你是对的。毕竟，我们将整个句子转换成一组原始的单词，忽略了句法、语法或者词序。这种转换明显丢失了信息。例如，“I like carrots and hate broccoli”（我喜欢胡萝卜，讨厌花椰菜）和“I like broccoli and hate carrots”（我喜欢花椰菜，讨厌胡萝卜）这两个句子转换成同一组标记[and; broccoli; carrots; hate; i; like]，因此被认为是完全一样的。与此同时，我们没有尝试理解句子的含义，只需要识别与特定行为有联系的词语。

我要强调的另一点是，虽然我们的标记化看起来相当简单，但是已经包含了许多隐含的决策。我们决定将“Programming”和“programming”当成同一个词，忽略大小写。这是个合理的假设吗？可能是，也可能不是。如果你曾经参加过在线讨论（如论坛或者 YouTube 评论），可能同意大量使用大写字母往往代表着讨论质量正在走下坡路。因此，这证明大小写很重要，可能提供分类消息的相关信息。另一个隐含的决定是丢弃“!”，这合理吗？标点也可能很重要。

好消息是，我们无须争论大小写或者标点符号是否重要，可以很容易地在以后的交叉验证中得到这个问题的答案。正如 W. Edwards Deming 的不朽名言：“我们信仰上帝——其他人都必须用数据证明自己。”为每个假设创建不同的模型，比较它们的表现，让数据做出决定。

2.3.2 简单组合得分

现在我们已经决定将短信分解为 3 个标志——“driving”“cant”和“txt”——仍然必须想出计算消息是非垃圾短信或垃圾短信概率的方法。“txt”表明有很大的概率是垃圾短信，而“driving”则指向非垃圾短信。我们如何组合所有证据？如果应用贝叶斯定理，确定在包含全部 3 个标记的条件下，某条短信是垃圾短信的概率，可以得到如下算式：

$$P(\text{SMS是垃圾短信}\mid\text{SMS包含driving、cant和txt})$$
$$=\frac{P(\text{SMS包含driving、cant和txt}\mid\text{SMS是垃圾短信})\times P(\text{SMS是垃圾短信})}{P(\text{SMS包含driving、cant和txt})}$$

这个公式看上去有些可怕。但是，如果我们做一个简化的假定，事情就容易多了。我们假定这些标记相互独立——也就是说，在一条短信中看到一个标记对其他标记是否出现没有任何影响。在这种情况下，计算公式是：

$$\begin{aligned}P(&\text{SMS 包含 driving、cant、txt}\mid\text{SMS 是垃圾短信})\\&=P(\text{SMS 包含“driving”}\mid\text{SMS 是垃圾短信})\\&\times P(\text{SMS 包含“cant”}\mid\text{SMS 是垃圾短信})\\&\times P(\text{SMS 包含“txt”}\mid\text{SMS 是垃圾短信})\end{aligned}$$

对以上公式，借用一点点贝叶斯定理的“柔道”翻身之术，可以得到如下结果：

$$\begin{aligned}P(&\text{SMS 是垃圾短信}\mid\text{SMS 包含 driving、cant 和 txt})\\&=P(\text{SMS 包含“driving”}\mid\text{SMS 是垃圾短信})\\&\times P(\text{SMS 包含“cant”}\mid\text{SMS 是垃圾短信})\\&\times P(\text{SMS 包含“txt”}\mid\text{SMS 是垃圾短信})\times\frac{P(\text{SMS是垃圾短信})}{P(\text{SMS包含driving、cant和txt})}\end{aligned}$$

从根本上，我们做了如下工作：没有尝试建立完整、复杂的英语模型，而是使用简单得多的模型。想象一下，你有两个装单词的大桶，一个用于垃圾短信，另一个用于非垃圾短信，它们包含的单词所占比例不同。消息通过从其中一个桶里随机选择单词组合生成。当然，这是相当可笑的语言模型。与此同时，该模型也有一些明显的好处。更“正确”的模式可能专用于特定语言（例如需要考虑其语法和句法），因而有可能对其他语言无效。因此，从较差但对任何语言或者文档类型效果相似且容易处理的模型开始，看看它会将我们引向何方！

2.3.3　简化的文档得分

现在，我们几乎已经为编码实现分类器做好了准备。我们可以轻松求取 P（SMS 是垃圾短信）——训练集中垃圾短信的比例，以及 P（SMS 包含“X”|SMS 是垃圾短信）——包含标记“X”的垃圾短信比例。

投入工作之前，先进行一些最后的调整。首先，你可能已经注意到，在垃圾短信和非垃圾短信中，冗长的贝叶斯公式都包括同一除数项——P（SMS 包含“driving”“cant”“txt”）的运算。最终，我们感兴趣的是决定一条消息是垃圾短信还是非垃圾短信，而不是精确的概率。在这种情况下，我们可以去掉公共项，节约不必要的计算，简单地计算“得分”而非概率。

Score (SMS 是垃圾短信 | SMS 包含 driving、cant 和 txt)
=*P* (SMS 包含“driving” | SMS 是垃圾短信)
×*P* (SMS 包含“cant” | SMS 是垃圾短信)
×*P* (SMS 包含“txt” | SMS 是垃圾短信) × *P*（SMS 是垃圾短信 ）

如果 Score（垃圾短信）>Scrore（非垃圾短信），我们将把消息归类为垃圾短信。

实际上，如果我们仅需要一个得分，可以进一步解决另一个问题——与精度相关的问题。从定义上看，从短信中任何特定标记上观察到的概率都是小于 1 的数字（通常接近 0）。由于我们的公式包含这些概率的乘积，结果也接近 0，可能造成舍入误差。

为了避免这一问题，习惯上是使用一种旧的技巧，将计算转换为对数。这方面的细节不是很重要，但正是使用这种方法的原因。因为 log (*a* * *b*) = log *a* + log *b*，可以将公式从乘积转换成总和，避免了舍入问题。而且，因为对数是递增函数，通过对数转换公式将保留得分的排名。所以，我们使用如下公式代替原公式，求得消息的分数：

Score (SMS 是垃圾短信 | SMS 包含 driving、cant 和 txt)=log (*P*(SMS 是垃圾短信))+
log（*Laplace* (SMS 包含“driving” | 垃圾短信)）+
log（*Laplace* (SMS 包含“cant” | 垃圾短信)）+
log（*Laplace* (SMS 包含“txt” | 垃圾短信))

在一定程度上，上式澄清了算法逻辑。每个标记都有一个非垃圾短信得分和垃圾短信得分，量化了它属于这两种情况的强度。尝试决定某个文档是不是垃圾短信时，算法首先计算垃圾短信得分——垃圾短信基准水平，每个标记都增加或者减少文档的垃圾短信得分，独立于其他标记。如果文档的垃圾短信得分最终高于非垃圾短信得分，则判定文档是垃圾短信。

2.4 实现分类器

我们已经讨论了许多数学和模型方面的内容，但是还没有讨论编码。幸运的是，这就是我们所需要的一切：现在，我们已经为实现一个简单的贝叶斯分类器做好了准备。从根本上说，分类器依赖于两个要素：将文档分解为标记的标记化程序，以及一组标记——用于求得文档得分的单词。有了这两个组件，我们需要从一个示例样本中知道每个标记的垃圾短信和非垃圾短信得分，以及每一组的相对权重。

图 2-4 概述了学习阶段。从具备标签的消息语料库开始，将它们分解为两组：垃圾短信和非垃圾短信，并计量其相对规模。然后，对选中的一组标记（“free”“txt”和“car”），计量其频度，将每一组文档归纳为对应于其总体权重的得分，为每个标记求得特定组别的得分。

得到上述信息之后，新文档的分类遵循图 2-5 中概述的过程：标记化文档，根据存在的标记计算每组得分，并预测最高得分的组。

在这个特殊的例子中，一旦文档标记化，我们搜索每个单独标记（忽略其他情况）是否都有一个得分，计算每组的总体得分，确定哪一组更可能匹配。

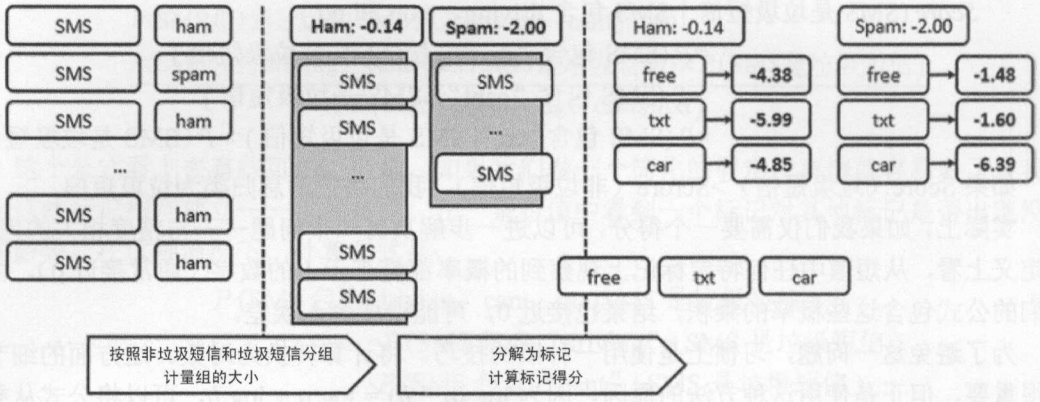


图 2-4　简单贝叶斯学习阶段

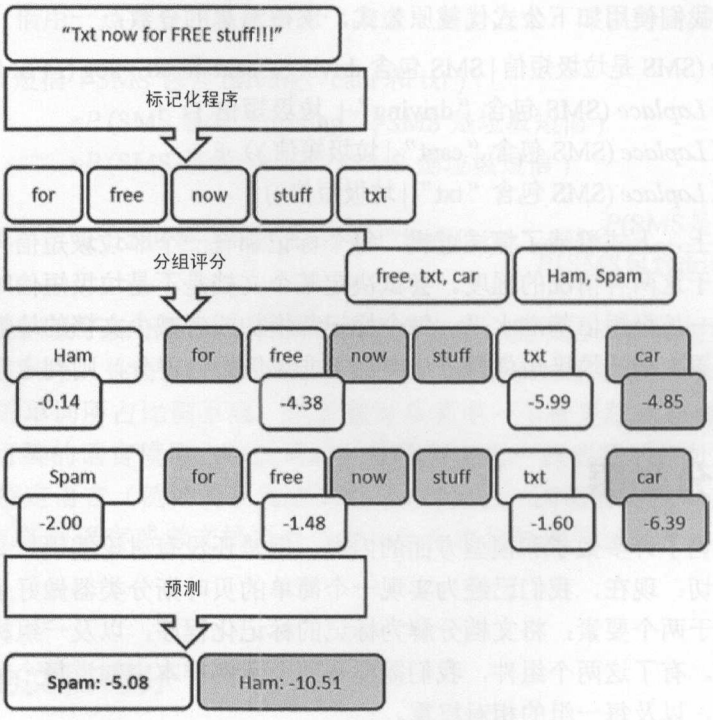


图 2-5　简单贝叶斯分类器概况

2.4.1　将代码提取到模块中

为了避免在脚本中聚集过多代码，我们将分类器提取为一个模块。这只需用鼠标右键单击解决方案，选择“新建项”（Add New Item），然后选择“源文件”（Source File）。将文件改名为 NaiveBayes.fs，删除其中的所有默认代码，用如下代码代替：

```
namespace NaiveBayes

module Classifier =

    let Hello name = printfn "Hello, %s" name
```

最后，用鼠标右键单击该文件并选择“上移”（Move up）（或者 Alt+向上箭头），直到NaiveBayes.fs 出现在解决方案文件列表中的第一个（也可以在解决方案资源管理器中选择文件，使用“在上方添加”（Add Above）或者“在下方添加”（Add Below）将文件直接添加到所需的位置）。现在，你已经创建了一个文档，可以开始编写函数了，从脚本中调用，或者从F#或者 C#项目中使用。为了说明，下面是从当前脚本使用 Hello 函数的方法：

```
#load "NaiveBayes.fs"
open NaiveBayes.Classifier
Hello "World"
```

注意：你可能觉得疑惑，为什么必须将刚创建的文件上移。和 C#不同，F#解决方案中的文件顺序很重要，主要原因是类型推理系统。编译器查看到目前为止代码文件中已有的定义，自动理解类型的含义。类似地，如果打算在代码中使用函数 *f*，该函数必须在使用它的代码之前定义。这是一个附加的限制，但是在我看来，相对于类型推理的好处，这种代价非常合理。而且，作为有趣的副作用，这提供了 F#解决方案中的自然顺序，使它们容易解读：从第一行开始正向阅读，或者从最后一行开始反向阅读！

2.4.2 文档评分与分类

有了模块，我们就可以在模块中写入上述算法，然后使用脚本文件以该算法探索数据。我们将遵循典型的 F#模式，由底向上构建，编写小的代码块组成更大的工作流。模型的关键要素是得分计算，这个功能计量文档属于某一组（如垃圾短信）的证据强度。

得分取决于两个成分：该组在整个语料库（训练数据）中出现的频度，以及该组中找到某些标记的频度。我们首先定义表现问题域的两个类型。我们不将文档和标记定义为字符串，而是直呼其名，定义一个 Token 类型——字符串的类型别名。这将使我们在类型签名上更加清晰。例如，现在可以定义一个 Tokenizer（标记化程序）函数，以一个字符串（文档）为参数，返回一组标记。类似地，使用 TokenizedDoc 类型别名，为已经标记化的文档提供一个名称：

```
type Token = string
type Tokenizer = string -> Token Set
type TokenizedDoc = Token Set
```

为了分类一个新文档（标签未知），我们需要使用已有的有标签文档样本，计算它在每个可能组的得分。这样做需要每个组的两部分信息：比例（该组在整个数据集中被找到的频率）以及为模型选择的每个标记的得分，这表明了标记指向该组的强度。

我们可以用一个数据结构 DocsGroup 建立模型，它包含 Proportion（该组在整个数据集中被找到的频率）和 TokenFrequencies——将每个标记与数据集中频率（用拉普拉斯方法校正）关联的 Map 类型（与字典作用类似的 F#数据结构）。举个例子，在本书的特例下，可以创建两个 DocsGroup——一个用于非垃圾短信，另一个用于垃圾短信——非垃圾短信 DocsGroup 包含整个数据集中非垃圾短信的比例，对于我们感兴趣的每个标记，还包含实际标记和出现在非垃圾短信中的频率。

```
type DocsGroup =
    { Proportion:float;
      TokenFrequencies:Map<Token,float> }
```

假定某个文档已经分解为一组标记，计算给定组的得分相当简单。我们没有必要担心这些数字的计算方法。如果这一部分已经完成，需要做的就是加总分组频率的对数，以及每个标记出现在已标记化文档中和模型使用的标记列表中频率的对数：

程序清单 2-2　计算文档得分

```
let tokenScore (group:DocsGroup) (token:Token) =
    if group.TokenFrequencies.ContainsKey token
    then log group.TokenFrequencies.[token]
    else 0.0

let score (document:TokenizedDoc) (group:DocsGroup) =
    let scoreToken = tokenScore group
    log group.Proportion +
    (document |> Seq.sumBy scoreToken)
```

然后，文档分类就只是用标记化程序将其转换为标记，找出可能的 DocGroups 列表中具有最高得分的分组，并返回其标签：

程序清单 2-3　预测文档标签

```
let classify (groups:(_*DocsGroup)[])
             (tokenizer:Tokenizer)
             (txt:string) =
    let tokenized = tokenizer txt
    groups
    |> Array.maxBy (fun (label,group) ->
        score tokenized group)
    |> fst
```

你可能对 classify 函数中的 groups:(_*DocsGroup)[]有些疑惑，为什么对每组文档使用

(DocType*DocsGroup)元组，神秘的_符号是什么？请你仔细思考，到目前为止，我们所做的并不依赖于具体的标签。我们考虑了垃圾短信和非垃圾短信，但是同样可以将其应用到不同语料库，例如，预测电影评论是否对应于 1、2、3、4 或者 5 星评级。因此，我决定使这些代码通用；只要有一组具备一致标签类型的示例文档，这些代码就有效。_符号表示“类型通配符”（任何类型都得到支持），如果检查 classify 函数签名，就会看到：groups:('a * DocsGroup) [] -> tokenizer:Tokenizer -> txt:string -> 'a。通配符已经被'a（表示泛型类型）所代替，函数也返回一个'a，这是（泛型）标签类型。

2.4.3 集合和序列简介

这里已经引入了两个新类型——序列和集合。我们从集合开始简短地讨论这两种类型，集合代表一组唯一项目，主要目的是回答“项目 X 是否属于集合 S”的问题。考虑到我们想要搜索特定单词（更通用的说法是标记）是否出现在 SMS 中，以确定 SMS 可能是垃圾短信还是非垃圾短信，使用可以有效比较项目集合的数据结构似乎很合理。通过将文档归纳为几组标记，就可以快速识别是否包含某个标记（如“txt”），而无须付出使用 Contains()字符串方法的代价。

Set 模块提供了许多围绕集合运算的方便函数，如下面的片段所示。在 F# Interactive 窗口中输入以下内容：

```
> let set1 = set [1;2;3]
let set2 = set [1;3;3;5];;

val set1 : Set<int> = set [1; 2; 3]
val set2 : Set<int> = set [1; 3; 5]
```

这段代码创建了两个集合，注意从 set2 删除重复的值 3 的方法。Set 模块提供了可以计算并集及交集的函数：

```
> let intersection = Set.intersect set1 set2
let union = Set.union set1 set2;;

val intersection : Set<int> = set [1; 3]
val union : Set<int> = set [1; 2; 3; 5]
```

集合还输出一个有用的函数 difference，从一个集合中减去另一个集合，也就是说，从第一个集合中删除第二个函数中的每个元素：

```
> let diff1 = Set.difference set1 set2
let diff2 = Set.difference set2 set1;;

val diff1 : Set<int> = set [2]
val diff2 : Set<int> = set [5]
```

注意，与 intersection 和 union 函数不同，difference 函数不可交换。也就是说，参数的顺序很重要，前一个例子中已经做了说明。最后要注意，集合是不可变的，一旦创建集合，它就不能修改：

```
> let set3 = Set.add 4 set1
set1.Contains 4;;

val set3 : Set<int> = set [1; 2; 3; 4]
val it : bool = false
```

在 set1 中添加 4 创建一个新集合 set3，它包含 4，但是原始集合 set1 没有受到这一运算的影响。

我们引入的另一个类型——序列，是惰性求值的元素序列——也就是说，它是仅在必要时计算的集合，这样可以潜在地减少内存或者计算使用量。在 F# Interactive 中尝试如下代码：

```
> let arr1 = [| for x in 1 .. 10 -> x |]
let seq1 = seq { for x in 1 .. 10 -> x };;

val arr1 : int [] = [|1; 2; 3; 4; 5; 6; 7; 8; 9; 10|]
val seq1 : seq<int>
```

表达式 arr1 和 seq1 非常类似；它们都生成从 1 到 10 的整数。但是，数组立即显示全部内容——这是立刻求值的。相反，seq1 显示为整数的一个序列，但是没有进一步的细节。实际内容只在代码需要的时候才生成。下面的例子可以进一步证明这一点，它取得 seq1 并将每个元素加倍，构建一个新的序列。注意，我们在序列中增加了其他的作用，每当序列的一个元素生成，就打印输出已经映射的元素：

```
> let seq2 =
    seq1
    |> Seq.map (fun x ->
        printfn "mapping %i" x
        2 * x) ;;

val seq2 : seq<int>
```

同样，结果是一个尚无实际内容的序列，因为目前没有任何必要的内容。我们改编代码，要求得到前 3 个元素的总和：

```
> let firstThree =
    seq2
    |> Seq.take 3
    |> Seq.sum;;
mapping 1
mapping 2
mapping 3

val firstThree : int = 12
```

执行上述代码时，打印输出“mapping 1”“mapping 2”和“mapping 3”，说明序列取出了前 3 个元素，这是计算总和所必需的。但是，剩下 7 个元素仍然没有必要，现在仍然没有求值。

那么，在我们已经有数组这样完美的集合类型的情况下，为什么还要有序列呢？第一个好处是惰性求值。因为序列只在“必要”时求值，我们可以用它们生成无限序列。例如，生成数组完全不可能生成的某种集合。下面的片段提供了一个人为的例子：我们创建一个由 1 和−1 交替形成的无限序列。很显然，在内存中实例化这种无限序列是不可能的，但是既然定义了它，我们就可以使用，例如计算任意数量元素的总和：

```
> let infinite = Seq.initInfinite (fun i -> if i % 2 = 0 then 1 else -1)
let test = infinite |> Seq.take 1000000 |> Seq.sum;;

val infinite : seq<int>
val test : int = 0
```

F#序列常常得到使用的另一个原因是，它们等价于 C#的 IEnumerable，所有.NET 集合类型本身都可以当成序列处理，因此，Seq 模块提供的函数可以利用实际集合类型中没有出现的功能性，对其进行操纵。考虑下面的例子：

```
> let arr = [| 1;2;2;3;4;4;4|]
let counts =
    arr
    |> Seq.countBy (fun x -> x)
    |> Seq.toArray;;

val arr : int [] = [|1; 2; 2; 3; 4; 4; 4|]
val counts : (int * int) [] = [|(1, 1); (2, 2); (3, 1); (4, 3)|]
```

我们生成了一个整数数组，然后想要计算每个整数出现了多少次。虽然数组没有直接支持这一功能（至少，在 F# 4.0 正式交付之前没有），但是序列有此功能，所以我们在数组上调用 Seq.countBy，将结果转换为数组，以便触发序列求值。

> **提示**：虽然序列有许多好处（无他，谁不想只计算所需要的？），但是惰性求值也有缺点，包括增加了调试的复杂性。

2.4.4 从文档语料库中学习

为了使用 classify 函数，我们必须拥有每组文档的摘要数据。生成该摘要数据需要哪些数据？组的比例需要组中的文档数以及文档总数。对于选择用于分类器的每个标记，必须计算标记化文档包含该标记的比例（经过拉普拉斯修正）。

注意：不管使用何种语言或者风格，这样实现应用程序设计都很有效：创建具备单一功能、具备清晰定义任务的小型组件，不用担心程序中其余部分发生的情况。这有助于提取具备最小数据结构的清晰模型。然后，你可以将所有组件连接起来，那时，改变设计（例如为了性能）就不会很困难。

收集摘要数据的明显方法是编写一个函数，签名中以文档集合为参数，返回包含传递文档组分析结果的 DocsGroup。我们仍然假定获得的数据形式对手上的任务很便利，如何实现这一点将在以后研究。

首先，创建两个助手函数，为我们完成比例和拉普拉斯平滑比例计算中的整数-浮点转换，帮助计算存在特定标记的文档数量：

```
let proportion count total = float count / float total
let laplace count total = float (count+1) / float (total+1)
let countIn (group:TokenizedDoc seq) (token:Token) =
    group
    |> Seq.filter (Set.contains token)
    |> Seq.length
```

现在，我们已经有了分析有相同标签的一组文件、然后将其归结为一个 DocsGroup 所需的所有组件，下一步需要做的就是：

- 计算该组与文档总数的比例。
- 对我们确定用于分类文档的每个标记，找出在组中的拉普拉斯修正比例。

程序清单 2-4　分析一组文档

```
let analyze (group:TokenizedDoc seq)
            (totalDocs:int)
            (classificationTokens:Token Set)=
    let groupSize = group |> Seq.length
    let score token =
        let count = countIn group token
        laplace count groupSize
    let scoredTokens =
        classificationTokens
        |> Set.map (fun token -> token, score token)
        |> Map.ofSeq
    let groupProportion = proportion groupSize totalDocs

    {
        Proportion = groupProportion
        TokenFrequencies = scoredTokens
    }
```

■ 注意：一定要计算标记的数量，我们将通过计算单词在多少文档中出现，确定其出现频率。这对于短消息非常合适，但是对更长的文档却不是很好。如果你试图识别一本书是关于何种主题（如计算机）的，计算这个单词出现的次数可能比简单地问“这个单词在本书中是否至少出现一次”更好。我们用于 SMS 的这种方法称为“单词集”；另一种标准方法“单词袋”取决于单词的整体频率，可能更适合于某些文档类型。

我们几乎已经完成工作了，现在已经知道如何分析单独的文档组，只需要编排这些代码块，完整地实现图 2-4 中概述的算法。对于文档及其标签的一个组合，我们需要：

- 将每个文档分解为标记。
- 按照标签分隔它们。
- 分析每个组。

程序清单 2-5 从文档中学习

```
let learn (docs:(_ * string)[])
          (tokenizer:Tokenizer)
          (classificationTokens:Token Set) =
    let total = docs.Length
    docs
    |> Array.map (fun (label,docs) -> label,tokenizer docs)
    |> Seq.groupBy fst
    |> Seq.map (fun (label,group) -> label,group |> Seq.map snd)
    |> Seq.map (fun (label,group) -> label,analyze group total classificationTokens)
    |> Seq.toArray
```

任务就要圆满完成了。从整个过程中，我们真正想得到的是一个函数，以字符串（原始 SMS 消息）为参数，根据从文档训练集中学到的，返回具有最高得分的标签。

程序清单 2-6 训练简单的贝叶斯分类器

```
let train (docs:(_ * string)[])
          (tokenizer:Tokenizer)
          (classificationTokens:Token Set) =
let groups = learn docs tokenizer classificationTokens
let classifier = classify groups tokenizer
classifier
```

程序大体就是如此，只花了大约 50 行代码，我们就得到了一个能够正常工作的简单贝叶斯分类器。

2.5 训练第一个分类器

实现通用算法之后，我们最终可以回到手上的问题——识别哪些消息是非垃圾短信，哪

些是垃圾短信。Train 函数的签名提供了目标的清晰概况：要获得分类器，需要一个示例训练集、一个标记化程序和选用的标记。我们已经得到了训练集，现在的目标是使用交叉验证指导分析，确定标记化程序和标记的最佳组合。

2.5.1　实现第一个标记化程序

考虑到上述情况，我们先完成可行的最简单工作，首先是标记化。我们所要做的是取得一个字符串，将其分解为单词，忽略大小写。

这项工作需要正则表达式：\w+模式匹配由一个或者多个“语言符号”（也就是字母或者数字，不包含标点符号）组成的“单词”。我们用该模式创建一个正则表达式 matchWords，在 script.fsx 中按照经典的管道结构编写一个 tokens 函数，将输入字符串转换为小写，并应用正则表达式。matchWords.Matches 创建应用表达式时找到的所有匹配项的集合 MatchCollection。我们将该集合转换为 Match 序列，读出匹配值（每个匹配的字符串），最后将其转换为一个包含正则表达式识别出的所有单词的集合。在脚本文件中现有的用于读取数据集的代码之后，添加如下代码：

程序清单 2-7　使用正则表达式标记化一行文本

```
open System.Text.RegularExpressions

let matchWords = Regex(@"\w+")

let tokens (text:string) =
    text.ToLowerInvariant()
    |> matchWords.Matches
    |> Seq.cast<Match>
    |> Seq.map (fun m -> m.Value)
    |> Set.ofSeq
```

这段代码看起来有些粗糙，说实话，如果不是因为.NET 正则表达式奇怪的设计特征，它看上去应该更简单。无论如何，我们已经有了所需要的东西：将字符串分解为所包含单词的函数。

2.5.2　交互式验证设计

初看之下，程序清单 2-7 中的代码似乎可以完成我们所需的功能。但是“似乎可以”还不能令人满意，更好的做法是验证实际工作情况。用 C#编码时，我的习惯是采用测试驱动开发（TDD）：先填写一个测试，然后编写满足要求、通过测试的代码。TDD 的好处之一是可以独立运行整个应用程序的小部分代码，快速确认它按照意图工作，然后逐步充实设计。在 F#中，我倾向于采用稍有不同的工作流。F# Interactive 可以实现更流畅的设计试验方式。你可以简单地在 F# Interactive 中试验，一旦设计成型，则提升代码进行单元测试，而不是立即提交设计并编写相关测试——那可能需要在以后重构。

在我们的例子中，检查 token 函数是否正常工作很简单。只需要在 FSI 中输入下面的语句并运行：

```
> tokens "42 is the Answer to the question";;
```

你将立刻在 F# Interactive 窗口中看到结果：

```
val it : Set<string> = set ["42"; "answer"; "is"; "question"; "the"; "to"]
```

我们的函数看起来工作得很好。所有单词已经分隔，包括数字“42”，重复的“the”在结果中只出现一次，“Answer”中的大写字母“A”也已经转换为小写。此时，如果我们编写的是用于生产环境的真实库，可以将 token 函数从探索性脚本中转移到解决方案中的相应文件，然后将这一个小代码片段转换为真正的单元测试。现在，我们“只是探索”，所以将保持原状，不马上编写测试。

2.5.3 用交叉验证确立基准

有了一个模型，我们就要观察它的运行状况。我们将要使用的脚本和以前一样，所以已经加载了数据集中的所有短信——（DocType * string）数组。和前一章中一样，我们将使用交叉验证，数据集的一部分用于训练，剩下的保留，用于验证每个模型效果好坏。我随意保留 1000 个观测值用于验证，剩下的观测值都用于训练。为了确认所有代码正常工作，我们尝试一个相当粗糙的模型，只使用标记“txt”进行决策：

```
#load "NaiveBayes.fs"
open NaiveBayes.Classifier

let txtClassifier = train training wordTokenizer (["txt"] |> set)

validation
|> Seq.averageBy (fun (docType,sms) ->
    if docType = txtClassifier sms then 1.0 else 0.0)
|> printfn "Based on 'txt', correctly classified: %.3f"
```

在我的机器上得出的正确率为 87.7%。这已经很不错了！不是吗？如果没有背景信息，87.7%这样的数字似乎已经很令人满意，但是必须正确看待。如果我使用最蠢的模型，不管 SMS 的内容为何都始终预测“非垃圾邮件”，在验证集上的成功率也可以达到 84.8%，因此 87.7%的成绩没有什么了不起。

我们所要强调的重点是，应该始终从确立基准开始：如果使用可能的最简策略，预测的效果如何？对于分类问题，始终预测频率最高的标签（正如上面所做的）可以提供一个出发点。对于时间序列模型（根据过去的情况预测明天），预测明天的情况和今天一样是经典的做法。在一些情况下，20%的分类正确率可能已经很出色了，而在其他情况下（例如非垃圾短信和垃圾短信的例子），85%的正确率都令人失望。基准完全取决于数据。

第二个重点是，应该很小心地对待数据集的构建方式。在分割训练和验证集时，我简单地使用前 1000 个数据项作为验证集。想想看，我的数据集按照标签排序，所有垃圾短信示例在前，然后是所有非垃圾短信。这很糟糕：我们实际上在几乎只包含非垃圾短信的样本上训练分类器，只在垃圾短信上进行测试。你最起码可以预计到，评估质量将会相当差，因为分类器学习识别所用的数据与实际处理的数据大不相同。

对于所有模型，这都是个问题。然而，在我们的例子中因为简单贝叶斯分类器的工作方式，这个问题特别严重。回忆一下算法的工作原理，本质上，简单贝叶斯算法以两个因素确定权重：标记在不同组的相对频率，以及组本身的频率。质量不佳的采样不会影响第一部分，但是，每个组的频率可能完全脱离实际，这将大大影响预测。例如，训练集中非垃圾短信多于垃圾短信，将导致分类器完全低估消息是垃圾短信的可能性，偏向于预测非垃圾短信，除非有占据压倒性优势的证据。

重点是，在运行可能很冗长的模型训练计算之前，必须问自己两个问题：“我使用的样本是否与模型以后所要处理的真实数据类似？”以及“我的训练集和验证集组成是否基本相同？”在某些情况下，使用失真的样本可能完全没有问题——你只需要意识到这一点即可。对于简单贝叶斯分类器或者基于贝叶斯定理的方法，特别要注意这类失真，它们对模型的质量将有深远的影响。如果检查 SMS 数据集，应该会发现它看上去大体是平衡的：在训练和验证样本中，非垃圾短信和垃圾短信的比例大约相同。

2.6　改进分类器

现在我们已经有了基准—— 低于 84.4%是糟糕的结果，至少要超过 87.7%。是时候观察使用可以自由支配的两大手段（标记化程序和分类器选用的标记）能达到什么效果了。

2.6.1　使用每个单词

利用一个单词，我们就能将分类器的正确率从 84.8%提升到 87.7%。如果使用每个可用的标记代替单个词汇，预测正确率当然应该大幅提高。我们来尝试一下这个思路，首先，必须从训练集中提取每个标记。我们编写一个 vocabulary 函数，对每个文档应用标记化程序，将标记合并为单一集合，实现上述功能。然后，从训练集中读取标记——用 snd 取出每个文档元组中的第二个元素，通过管道进入 vocabulary 函数：

```
let vocabulary (tokenizer:Tokenizer) (corpus:string seq) =
    corpus
    |> Seq.map tokenizer
    |> Set.unionMany

let allTokens =
```

```
    training
    |> Seq.map snd
    |> vocabulary wordTokenizer
```

干脆利索。现在我们可以使用大的标记列表训练新的分类器，评估其性能：

```
let fullClassifier = train training wordTokenizer allTokens

validation
|> Seq.averageBy (fun (docType,sms) ->
    if docType = fullClassifier sms then 1.0 else 0.0)
|> printfn "Based on all tokens, correctly classified: %.3f"
```

结果相当令人失望，没有出现我们希望的重大改善："根据所有标记，正确分类的比例为 0.864"。我们必须面对这一失败，得到的模型仅仅胜过最简单的预测程序，比依赖单一词汇"txt"决策的前一个模型更差。

在此得到的教训是，盲目地向一个标准算法中投入许多数据，只会离目标越来越远。在我们的例子中，精心选择了一个特征，形成的模型更简单、更快、更擅长于预测。选择合适的特征是构建好的预测程序的关键部分之一，也是我们将要尝试推进的。在本例中，可以在两方面改变特征的定义，而无须更改算法本身：可以使用不同的标记化程序，以不同方式利用消息，也可以选择（或者忽略）不同的标记集合。

忙于这些工作时，我们将会多次重复某些代码，可以将整个"训练—评估"过程打包为一个函数，该函数以一个标记化器和一组标记为参数，运行训练过程，打印输出评估结果：

程序清单 2-8　通用模型评估函数

```
let evaluate (tokenizer:Tokenizer) (tokens:Token Set) =
    let classifier = train training tokenizer tokens
    validation
    |> Seq.averageBy (fun (docType,sms) ->
        if docType = classifier sms then 1.0 else 0.0)
    |> printfn "Correctly classified: %.3f"
```

评估特定模型变成一行代码：

```
> evaluate wordTokenizer allTokens;;
Correctly classified: 0.864
```

这为我们提供了相对简单的推进过程：选择一个标记化器和一组标记，调用 evaluate 函数观察特定组合是否好于之前尝试的组合。

2.6.2　大小写是否重要？

我们从标记化程序开始，回忆之前的讨论，目前使用的 wordTokenizer 函数忽略大小写，从它的角度，"txt"和"TXT"之间没有差别——两者被视为同一个标记。

这并不是不合理的做法。例如，考虑消息“Txt me now”和“txt me now”（给我发文本消息），忽略大小写，我们实质上将这两条消息视为意义上没有任何差别的消息。大小写没有带来任何相关的信息，可以当成噪声。

相反，考虑这两条消息：“are you free now?”（你现在有空吗？）和“FREE STUFF TXT NOW”（这是有关免费物品的短信）——这时，忽略大小写就可能丢失信息。

那么，哪一种方法才对呢？我们来尝试一下，看看不将所有字符串转换为小写的不同标记化程序是否比 wordTokenizer 更好：

程序清单 2-9　使用正则表达式标记化一行文本

```
let casedTokenizer (text:string) =
    text
    |> matchWords.Matches
    |> Seq.cast<Match>
    |> Seq.map (fun m -> m.Value)
    |> Set.ofSeq

let casedTokens =
    training
    |> Seq.map snd
    |> vocabulary casedTokenizer

evaluate casedTokenizer casedTokens
```

在 FSI 中运行达到 87.5%的正确率。一方面，我们的结果仍然低于依赖“txt”的简单分类器；另一方面，两者已经相当接近（87.5%对 87.7%），这明显好于使用所有单一标记的 wordTokenizer（86.4%）。

2.6.3　简单就是美

我们得到了表面上更好的标记化程序，但是，添加大量特征所得到的结果仍然是效果更差、更缓慢的分类器。这明显不理想，也是反直觉的：增加更多信息怎么可能使模型更差？

让我们从另一方面去看待问题，思考之前的单标记模型为什么工作得更好。我们选择“txt”作为标记的原因是它在垃圾短信中经常发现，而在非垃圾短信中很少出现。换言之，它很好地区分两组，相当特定于垃圾短信。

我们现在做的是使用每个单一标记，不管它们的信息量如何。结果是，我们在模型中可能引入了相当多的噪声。更有选择性的方法——可能只选择每个文档组中最常见的标记，会得到什么样的结果呢？

让我们从一个简单的函数开始，给定一组原始文档（简单字符串）和一个标记化程序，将返回最常用的标记：

```
let top n (tokenizer:Tokenizer) (docs:string []) =
    let tokenized = docs |> Array.map tokenizer
    let tokens = tokenized |> Set.unionMany
    tokens
    |> Seq.sortBy (fun t -> - countIn tokenized t)
    |> Seq.take n
    |> Set.ofSeq
```

现在，我们可以将训练样本分拆为非垃圾短信（Ham）和垃圾短信（Spam），在此过程中去掉标签：

```
let ham,spam =
    let rawHam,rawSpam =
        training
        |> Array.partition (fun (lbl,_) -> lbl=Ham)
    rawHam |> Array.map snd,
    rawSpam |> Array.map snd
```

我们提取和计数每个组的标签，提取前 10%，用 Set.union 将其合并为一个标记集合：

```
let hamCount = ham |> vocabulary casedTokenizer |> Set.count
let spamCount = spam |> vocabulary casedTokenizer |> Set.count

let topHam = ham |> top (hamCount / 10) casedTokenizer
let topSpam = spam |> top (spamCount / 10) casedTokenizer

let topTokens = Set.union topHam topSpam
```

现在，可以将这些标记整合为一个新的模型，评估其表现：

```
> evaluate casedTokenizer topTokens;;
Correctly classified: 0.952
```

正确率从 87.5%一跃达到 95.2%!这是可观的改善，实现这一飞跃不是通过增加更多的数据，而是通过删除特征。

2.6.4 仔细选择单词

我们还能做得更好吗？试试看！通过选择标记的一个子集，只保留携带有意义信息的标记，我们得到了很明显的改善。按照这一思路，可以开始检查非垃圾短信和垃圾短信中最常用的标记，这相当容易实现：

```
ham |> top 20 casedTokenizer |> Seq.iter (printfn "%s")
spam |> top 20 casedTokenizer |> Seq.iter (printfn "%s")
```

运行上述语句产生表 2-1 中的结果：

表 2-1　　最常用的非垃圾短信和垃圾短信标记

分组	标记
非垃圾短信	I, a, and, for, i, in, is, it, m, me, my, not, of, on, s, that, the, to, u, you
垃圾短信	1, 2, 4, Call, FREE, a, and, call, for, from, have, is, mobile, now, on, or, the, to, you, your

我们立刻注意到两件事。首先，两个列表都很相似，在 20 个标记中，有 8 个是共同的（a、and、for、is、on、the、to、you）。而且，这些单词都非常通用，不管什么主题，任何语句都必然包含几个冠词和代词。这一列表并不重要，表中的标记可以描述为“填充材料”，它们没有告诉我们很多关于消息是非垃圾短信还是垃圾短信的情况，甚至在我们的分析中引入了一些噪声。从性能的角度看，这意味着我们花费了许多 CPU 周期分析传达有限甚至毫不相关信息的标记。因此，从分析中去掉这些标记可能是有益的。

在此之前，我想到了列表中出现的另一个有趣特征。和非垃圾短信不同，垃圾短信的列表中包含几个相当特殊的单词——如“call”“free”“mobile”和“now”，不包含“I”或者“me”。如果你考虑到这一点，那是很有意义的：常规的文本消息有许多种目的，而垃圾短信通常试图“钓鱼”，诱惑你做某些事情。在这样的背景下，看到“now”（现在）和“free”（免费）这些带着广告腔调的单词也就不足为奇了，因为通过奉承的方法达到目的通常没有错，垃圾短信很少和“I”（我）相关而更多地与“You”（你）相关也是正常的。

注意：什么是“钓鱼”？钓鱼是使用电子邮件、SMS 或者电话从某人那里窃取财物的行为。钓鱼通常采用一定的社会工程（伪装成受尊重的机构、威胁或者引诱）使目标采取骗子无法独自完成的措施，比如单击一个在机器上安装恶意软件的链接。

我们将很快回到上述要点。与此同时，观察一下，能否删除“填充材料”。常见的方法是依赖这些单词的预定义列表，这通常称作“停用词”。在 http://norm.al/2009/04/14/list-of-english-stop-words/上可以找到一个此类列表。遗憾的是，没有通用的正确方法，显然，这取决于编写文档所用的语言，甚至语境也很重要。例如，在我们的例子中，文本消息中常用的“you”缩写词“u”是一个很合适的“停用词”。但是，大部分标准列表不会包含它，因为这种用法与 SMS 的关联度极大，不太可能在其他文档类型中找到。

我们不依赖于停用词列表，而是采用更简单的方法。我们的 topHam 和 topSpam 集合中包含非垃圾短信和垃圾短信中最常用的标记。如果某个标记在两个列表中都出现，它很有可能就是在英语文本消息中常见的单词，不特定于非垃圾短信和垃圾短信。我们找出所有共有的标记（这对应于两个列表的交集），将它们从标记选择中删除，再次运行分析：

```
let commonTokens = Set.intersect topHam topSpam
let specificTokens = Set.difference topTokens commonTokens
evaluate casedTokenizer specificTokens
```

利用这一简单的更改，分类 SMS 消息的正确率从 95.2%上升到了 97.9%。这可以认为是相当好的结果了，离 100%越近，改善越难实现。

2.6.5 创建新特征

在尝试提出新想法时，我常常觉得有用的方法之一是颠覆问题，以便从不同角度观察。举个例子，当我编写代码时，通常从想象“快乐路径”——程序完成某项功能所需的最少步骤——开始，然后实现它们。在测试代码之前，这是很棒的方法，你的心思已经完全集中在成功路径上，而难以考虑失败的情况。所以，我喜欢在测试时将问题颠倒并提问：“使这个程序崩溃的最快方法是什么？”我发现这一招非常有效，而且使测试变得很有趣！

让我们来试试用这个方法改进分类器。对每个类别中最常见单词进行观察最终可以得到很明显的改善，因为贝叶斯分类器依赖于分类中的常见词以识别类别。观察一下每个文档类别中最少见的词如何？稍微修改一下前面编写的代码，就可以提取分组中最少使用的标记：

程序清单 2-10　提取最少使用的标记

```
let rareTokens n (tokenizer:Tokenizer) (docs:string []) =
    let tokenized = docs |> Array.map tokenizer
    let tokens = tokenized |> Set.unionMany
    tokens
    |> Seq.sortBy (fun t -> countIn tokenized t)
    |> Seq.take n
    |> Set.ofSeq

let rareHam = ham |> rareTokens 50 casedTokenizer |> Seq.iter (printfn "%s")
let rareSpam = spam |> rareTokens 50 casedTokenizer |> Seq.iter (printfn "%s")
```

表 2-2 总结了结果：

表 2-2　最少使用的非垃圾短信和垃圾短信标记

分组	标记
非垃圾短信	000pes, 0quit, 100, 1120, 116, 1205, 128, 130, 140, 15pm, 16, 180, 1Apple, 1Cup, 1IM, 1Lemon, 1Tulsi, 1mega, 1stone, 1thing, 2000, 21, 21st, 24th, 255, 26, 2B, 2DOCD, 2GETHA, 2GEVA, 2Hook, 2I, 2MOROW, 2MORRO, 2MWEN, 2WATERSHD, 2bold, 2geva, 2hrs, 2morow, 2morrowxxxx, 2mro, 2mrw, 2nhite, 2nite, 2u, 2u2, 2years, 2yrs, 30ish
垃圾短信	0089, 0121, 01223585236, 0207, 02072069400, 02085076972, 021, 0430, 050703, 0578, 07, 07008009200, 07090201529, 07090298926, 07099833605, 07123456789, 07742676969, 07781482378, 077xxx, 078, 07801543489, 07808247860, 07808726822, 07821230901, 078498, 0789xxxxxxx, 0796XXXXXX, 07973788240, 07XXXXXXXXX, 08, 08000407165, 08000938767, 08002888812, 08002986030, 08081263000, 0825, 083, 0844, 08448350055, 08448714184, 0845, 08450542832, 08452810075over18, 08700435505150p, 08700469649, 08700621170150p, 08701213186, 08701237397, 0870141701216, 087016248

你是否注意到某种模式？垃圾短信列表中充满了电话号码或者文本编码。同样，这也很有意义：如果我成为“钓鱼”的目标，有人希望我做某件事情——在电话上，这意味着拨打某个号码或者用短信发送一个数字。

这也强调了一个问题：作为人类，我们立刻发现这个列表是“许多电话号码”，但是每个号码在模型中都被视为单独的标记，这些标记出现的很少。然而，SMS 消息中存在电话号码似乎是垃圾短信的可能标记。我们可以创建一个新特征，捕捉消息是否包含电话号码，而不管这些号码是什么，解决这个问题。这将使我们能够计算电话号码在垃圾短信中出现的频率，并（潜在地）在我们的简单贝叶斯分类器中将其作为一个标记使用。

如果仔细研究少见标记列表，就有可能发现更特殊的模式。首先，列出的电话号码都有类似的结构：它们以 07、08 或者 09 开头，然后是 9 个其他号码。其次，列表中有其他一些数字，主要是 5 位数字，这很可能是短信编码。

我们为每种情况创建一个特征。每当看到 07 之后的 9 个数字，就将其转换为标记__PHONE__，每当遇到 5 位数字，就将其转换为__TXT__：

程序清单 2-11　识别电话号码

```
let phoneWords = Regex(@"0[7-9]\d{9}")
let phone (text:string) =
    match (phoneWords.IsMatch text) with
    | true -> "__PHONE__"
    | false -> text

let txtCode = Regex(@"\b\d{5}\b")
let txt (text:string) =
    match (txtCode.IsMatch text) with
    | true -> "__TXT__"
    | false -> text

let smartTokenizer = casedTokenizer >> Set.map phone >> Set.map txt
```

smartTokenizer 简单地将 3 个函数链接在一起。casedTokenizer 函数取得一个字符串并返回一个字符串集合，包含单独识别的标记。因为它返回的是字符串集合，可以应用 Set.map，对每个标记运行 phone 函数，将看起来像电话号码的标记转换为“__PHONE__”，然后同样执行 txt 函数。

在 F# Interactive 中运行如下代码，确认上述函数正常工作：

```
> smartTokenizer "hello World, call 08123456789 or txt 12345";;
val it : Set<string> =
  set ["World"; "__PHONE__"; "__TXT__"; "call"; "hello"; "or"; "txt"]
```

结果正如我们预期的那样——一切正常。在列表中人工添加刚刚创建的两个标记，尝试调整过后的标记化程序（如果不这么做，标记列表中仍然包含单独的数字，没有与“智能标记化程序”产生的__PHONE__标记匹配）：

```
let smartTokens =
    specificTokens
    |> Set.add "__TXT__"
```

```
    |> Set.add "__PHONE__"

evaluate smartTokenizer smartTokens
```

请击鼓喝彩……准确率又一次跃升，从 96.7%上升到 98.3%！考虑到目前的性能水平，这是非常了不起的提升，值得花一些时间讨论。我们转换了数据集，通过聚合现有特征创建了一个新特征。这一过程是开发好的机器学习模型的关键部分。从原始数据集（可能包含低质量数据）开始，随着对领域的理解的进一步深入，找出变量之间的关系，就可以不断将数据重塑为新的表现形式，更好地适应手上的问题。这种试验、将原始数据精炼为好的特征的循环是机器学习的核心。

2.6.6 处理数字值

让我们以一个简短的讨论结束关于特征提取的部分。在本章中，我们只考虑离散特征，也就是说，只取一组离散值的特征。对于连续特征（或者数值特征）该怎么办？例如，想象一下，如果你的直觉是消息的长度很重要，能否使用到目前为止已经有的想法？

可能性之一是将问题简化为已知的问题，将消息长度（0～140 个字符）变成一个二分特征，将其分为两类——短消息和长消息。这样，问题就变成了“区分长消息和短消息的合适长度值是多少？”

我们可以直接用贝叶斯定理解决这个问题：对于指定的阈值，我们可以计算长于该阈值的消息是垃圾短信的概率。然后，识别“好”的阈值就只是测试不同数值，选择信息量最大的一个。下面的代码完成上述功能，应该不太难以理解：

程序清单 2-12　根据短信长度计算是垃圾短信的概率

```
let lengthAnalysis len =

    let long (msg:string) = msg.Length > len

    let ham,spam =
        dataset
        |> Array.partition (fun (docType,_) -> docType = Ham)
    let spamAndLongCount =
        spam
        |> Array.filter (fun (_,sms) -> long sms)
        |> Array.length
let longCount =
        dataset
        |> Array.filter (fun (_,sms) -> long sms)
        |> Array.length

let pSpam = (float spam.Length) / (float dataset.Length)
let pLongIfSpam =
```

```
        float spamAndLongCount / float spam.Length
let pLong =
        float longCount /
        float (dataset.Length)

let pSpamIfLong = pLongIfSpam * pSpam / pLong
        pSpamIfLong


for l in 10 .. 10 .. 130 do
        printfn "P(Spam if Length > %i) = %.4f" l (lengthAnalysis l)
```

运行上述代码，将会看到一条短信是垃圾短信的概率随着长度的增长而显著增大，这并不奇怪。垃圾短信制造者发送许多消息，希望得到最合理的收益，因此会在单条 SMS 中加入尽可能多的内容。现在，我将暂时搁置这一话题，在第 6 章中从稍有不同的角度讨论，但是我希望说明的是，贝叶斯定理在许多不同的情况下都很方便，不需要花费太多的精力。

2.7　理解分类错误

记得在正确看待、与简单基准比较之前，我们还认为 87.7%的分类正确率很不错吗？我们可以将这个数字提升到 98.3%，明显比基准要好得多。但是，每当将整个数据集归纳为单一数字时，从定义上说肯定要遗漏某些信息。人们喜欢用单一数字总结，因为这样容易比较。我的建议是：每当得到一个统计数字时，应该仔细思考该数字可能遗漏或者隐藏的细节。

在我们的例子中，所知道的是在 1000 条消息中，平均会错误分类 17 条。我们所不知道的是模型的哪一部分造成了错误。同样的数字（98.3%）可能是不同情况造成的：这 17 条消息可能主要是垃圾短信、主要是非垃圾短信或者介于两者之间。根据语境的不同，可能会造成关键的差别。例如，我怀疑大部分人更喜欢忽略一两条垃圾短信、将其保留在收件箱中的分类器，而不喜欢过于激进、因为不正确地标记垃圾短信而将完全正常的消息自动放到垃圾短信文件夹中的分类器。

那么，我们的分类器表现如何？让我们来发现和计量其识别垃圾短信和非垃圾短信的优劣。这只需分离验证集中的垃圾短信和非垃圾短信，然后计算每组正确分类的百分比，很容易完成——只需要在脚本中添加几行，在 F# Interactive 中运行：

程序清单 2-13　按照类别区分的分类错误

```
let bestClassifier = train training smartTokenizer smartTokens
validation
|> Seq.filter (fun (docType,_) -> docType = Ham)
|> Seq.averageBy (fun (docType,sms) ->
        if docType = bestClassifier sms
        then 1.0
        else 0.0)
```

```
|> printfn "Properly classified Ham: %.5f"
validation
|> Seq.filter (fun (docType,_) -> docType = Spam)
|> Seq.averageBy (fun (docType,sms) ->
    if docType = bestClassifier sms
    then 1.0
    else 0.0)
|> printfn "Properly classified Spam: %.5f"
```

产生的输出如下：

```
Properly classified Ham: 0.98821
Properly classified Spam: 0.95395
```

很不错，我们的分类器在两个类别上都工作得很好，在非垃圾短信组上表现最好——只有1.2%的消息分类错误。这一数字被称作“假阳性率”——检测出实际上不存在的情况的比率。类似地，“假阴性率”是忽略某个问题的比例。在我们的例子中，大约 4.6%的垃圾短信未能检出。

那么，为什么说这些信息有用？原因有以下两点。

首先，深入了解分类错误，理解它们是假阳性还是假阴性，在评估分类器商业价值时极其重要。这种价值完全取决于环境，实质上取决于哪一类错误代价更高。因为一条消息被错误地分类为垃圾短信而错过与老板的会面，远比从收件箱中人工删除一条垃圾短信严重得多。在这种情况下，假阳性比假阴性代价大得多，不应该使用相同的权重。

深入了解错误还有助于我们将精力集中在正确的方向，避免在无意义的工作上浪费时间。如果我们的分类器已经达到 100%的非垃圾短信识别率，剩下的唯一改进方向就在另一个类别。出发点可能是检查当前模型没有识别为垃圾短信的消息，看看是否出现某种模式？

理想状态下，我们所希望的是即使付出遗漏某些垃圾短信的代价，也要尽可能可靠地识别非垃圾短信。例如，实现这一点的途径之一是“调低”垃圾短信的灵敏度，而在非垃圾短信上更努力尝试。应急的方法是，我们可以减少来自垃圾短信方向的标记，增大来自非垃圾短信方向的标记。换言之，只保持较为特殊的垃圾短信标记——强烈的指标，而张大捕捉非垃圾短信标记的网。当前最佳模型中为每一类保留 10%的最常见标记，作为替代，如果我们在非垃圾短信上保留 20%，在垃圾短信上保留 5%，将会得到什么样的结果？让我们来尝试一下，使用完全相同的代码，但是将 topHam 和 topSpam 修改为：

```
let topHam = ham |> top (hamCount / 5) casedTokenizer
let topSpam = spam |> top (spamCount / 20) casedTokenizer
```

利用上述代码得到的结果是：总体准确率从 98.3%下降到 97.4%，但是正确分类的非垃圾短信从 98.8%上升到 99%，垃圾短信从 95.4%下降到 92.8%。换句话说，我们以另一个类别为代价，获得了非垃圾短信上的小幅提升。0.2%的改进看起来可能不多，但是我们真正实现的是将错误率从每千条消息 12 次下降到 10 次，也就是在最重要的类别上实现了 16.7%的改进。

2.8　我们学到了什么?

我敢肯定，用这个模型还可以做得更多，毕竟，我们距离完美还有 1.7%!但是，我将把这个问题留给读者作为兴趣练习，而将篇幅留给本章关键点的回顾。

首先，我们讨论了贝叶斯定理和独立性的概念。我觉得这一成果特别有趣，原因至少有两个。首先，贝叶斯定理的通用公式相当简单。其次，它很强大：贝叶斯定理提供了描述和量化两部分不完善信息之间关系的通用框架。它本身就是统计学和机器学习的一个关键成果，因为这两个学科都试图从不完整的样本中得出结论，找出数据之间的模式。也就是说，找出不独立的数据并加以利用。

当然，简单贝叶斯分类器本身在贝叶斯定理应用中微不足道：我们在文本信息的工作方式上做了一些简化的假设，以便使数学方法可行，只是为了展示应用贝叶斯定理的结果。文本标记化的过程可能比分类器本身更有趣。机器学习算法通常需要一组特征，原始的文本块并不能很好地适应这些模式。通过将文本分解为标志，识别和计算出现在一个文档中的单词数量，我们就能将文本转换为更方便的结构。

最后，我们花费很多时间改进基本模型，通过提取新特征加以完善。关键的要点是，我们始终使用相当简单的算法，但是仅仅花费时间理解数据、使用所得到的知识稍作重整，就能将模型的预测能力从中等提高到很好的水平。决定使用哪一个算法无疑是机器学习的重要部分之一，但是构建一个好的模型通常不是寻找唯一算法、魔法般地解决问题，而更多的是凭借对问题域的更深入理解，使用这些知识提取更好的特征。

在这个特例中，出现了几个模式。首先，数据并不总是越多越好。有些特征包含许多信息，有些则不然。找出信息量大的特征、消除其余特征可以为模型提供携带信号较多、噪声较少、更容易利用的信息。然后，我们看到了两个相反的现象，最终使用了保持单词大小写的标记化程序。我们意识到，原来使用的特征过于粗糙，因此将它们分解为多个更具体的特征（例如，“FREE”和“free”）。相反，我们注意到有许多电话号码，将其聚合成单个之前不存在的首要特征，以便观察其作用。同样，了解问题域和数据绝对是很重要的：如果好的算法以错误的方式观察特征，也不会走得太远，对特征的小更改往往能将一个普通的模型转化为出色的模型。

实用链接

- 加州大学欧文分校维护了一个非常出色的机器学习数据集知识库：http://archive.ics.uci. edu/ml/。
- 下面的网页包含一个实用的常见英语停用词列表：http://norm.al/2009/04/14/list-of-english-stop-words/。
- 如果你打算进一步探索文本处理方法，斯坦福大学的 NLP 页面有丰富的资源：http://nlp.stanford.edu/。

第 3 章

类型提供程序的快乐

寻找和准备来自任何地方的数据

现在我带你了解机器学习的一个小秘密：如果你只注意公开讨论的主题，可能会认为大部分工作都是围绕设计奇特算法进行的，剩下的时间将花费在以分布方式运行这些算法，或者同样有趣的设计难题上。很遗憾，真相是，那只是工作中细支末节的部分，你的大部分时间可能花费在更乏味的活动上：数据管理工作。如果希望机器学习任何东西，就必须向其提供数据。数据可能来自不同来源，所采用的形式和格式可能并不适合于你的处理。这些数据通常会丢失信息，很少有合适的文档。简言之，寻找和准备数据对于机器学习极为重要，这可能是痛苦的根源之一。

除了数据本身状态不佳之外，管理数据很棘手还有另一个原因。这一原因是技术上的：想象一下，你所要做的是将信息（这些信息对你的编程语言一无所知）引入编程语言类型系统，以便用编程语言对其进行实际的操纵。这时可以选择两个宽泛的方向，例如，如果选用的语言是动态类型语言（如 Python），对数据进行改造相当容易。你可以乐观的方式编写代码，不用过多担心语言类别。如果代码有效，那就好极了！这样做的缺点是，你无法从编译器中得到任何帮助。知道代码是否有效的唯一方法是运行。如果没有观察到运行时错误，就可以假定代码可能有效。相反，如果使用 C#之类具有更严格类型系统的语言，将数据引入你的世界将更加痛苦、费时，可能需要“重型设备”，例如对象-关系映射系统（ORM）。不过，这种语言的好处是可以避免你犯下愚蠢的错误：如拼写错误或者明显的类型转换错误。

这是一种令人不快的妥协：你可以很快地获得数据，但是无法确定得到的是什么；或者得到稳固的数据，但是牺牲敏捷性。幸运的是，F#提供了一个真正有趣的选项，可以使用称作类型提供程序的机制，在不做出任何牺牲的情况下获得静态类型的所有好处。在本书编写的时候，其他语言都不存在类型提供程序。类型提供程序（Type Provider，缩写为 TP）是针对特定资源类型或者数据格式（如 SQL、XML 或者 JSON）设计的一个组件。可以简单地将

合适的类型提供程序指向感兴趣的数据源，它将检查数据源的模式并在运行中创建可以安全使用的静态类型表现形式，为你完成繁重的工作。

类型提供程序与机器学习并无太大的关联，但是数据访问和操纵是数据科学活动的核心部分，我想花些时间解释数据探索的一般问题。在本章中，我们将说明如何使用类型提供程序处理来自广泛的不同来源的数据。我们将引入 Deedle，这个.NET 数据框架库极大地简化了数据集的操纵和重整。最后，我们还将展示类型提供程序与机器学习直接相关的使用方法，也就是在 F#内部直接使用开源统计语言 R。

3.1　探索 StackOverflow 数据

对于编写过代码的大部分人来说，StackOverflow 无须介绍。但是，为了以防万一，在此赘述两句：StackOverflow 是 Jeff Atwood 和 Joel Spolsky 创建的网站，它很快就成为了提出编程问题的权威场所。随着时间的推移，一些类似的网站已经萌发，专注于不同的专业领域，从数学到英语用法或者家庭酿酒，这些网站团结在 StackOverflow 旗下（顺便说一下，Cross-Validated 网站（http://stats.stackexchange.com）是统计学和机器学习的绝佳资源）。StackOverflow 从一开始就采用非常开放、与维基百科类似的哲学，所以提供 StackOverflow 及其姐妹网站的开放 API，允许你编程挖掘所有数据也就毫不奇怪了。在本节中，我们将了解如何利用 F#类型提供程序，通过 API 探索可用数据。

3.1.1　StackExchange API

StackExchange API 的文档在 http://api.stackexchange.com/docs 上。这是一个 REST API，返回 JSON 响应。有些方法需要身份验证，使用量受到限制。如果想使用该 API 构建一个真正的应用程序，可以获得一个键码，但是对于轻量级的探索工作，使用量限制不成问题。可以直接在一个网页上尝试 API 提供的所有方法，帮助你理解参数的工作方式。例如，访问 http://api.stackexchange.com/docs/questions 可以通过日期或者标签等多种条件搜索 StackOverflow 问题。

图 3-1 展示了对 2014 年 1 月 1～2 日间 C#问题的搜索。每页包含 20 条查询结果，我们请求结果的第 2 页；也就是第 21～40 号问题。单击“Run”按钮，当前查询将执行，结果显示在同一个窗口的下方。

还要注意查询的编辑方法，“Run”按钮左侧的部分根据你的编辑操作而变化，这部分显示的是用于获取数据的实际查询。举个例子，如果你搜索标记为 C#的问题，页面上将出现图 3-2 中的页面。

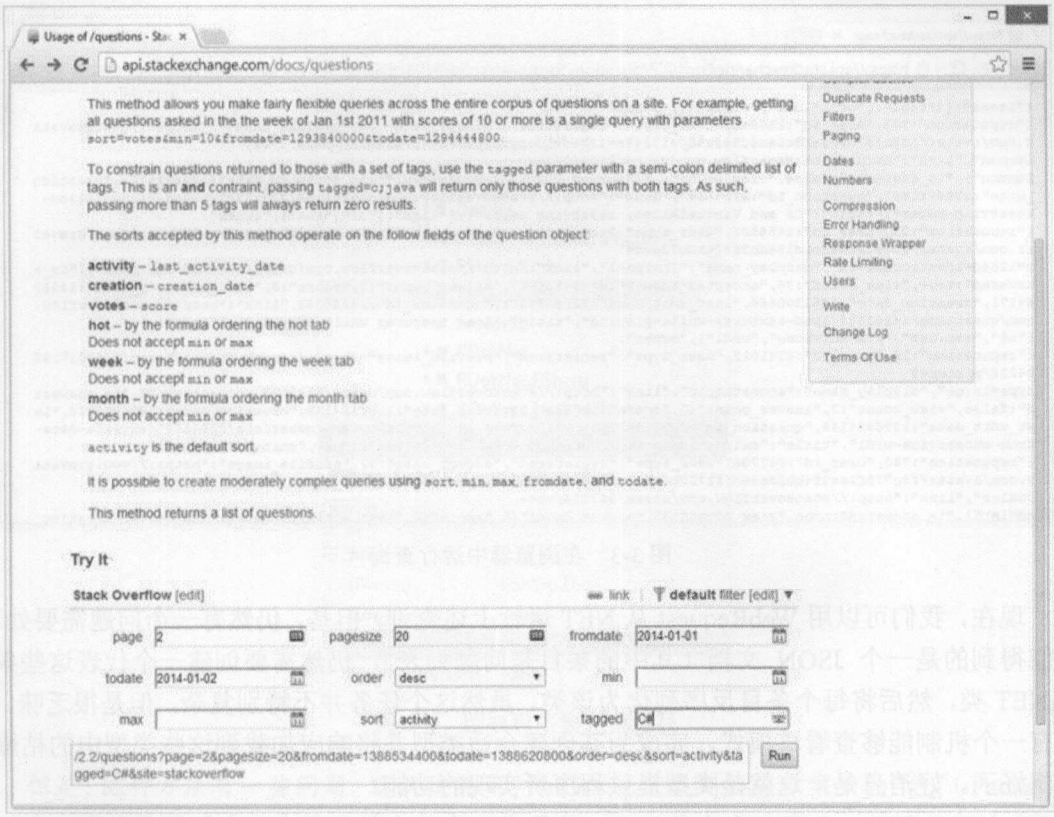


图 3-1 StackExchange API 的试用功能

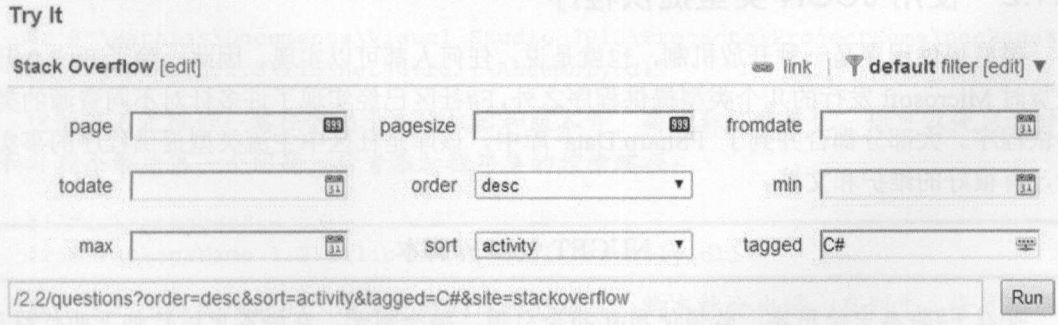


图 3-2 简单查询

现在，可以将查询结果粘贴到你的浏览器，在前面添加 http://api.stackexchange.com/。例如，https://api.stackexchange.com/2.2/questions?order=desc&sort=activity&site=stackoverflow&tagged=C%23（注意，我们用 C%23 代替 C#，用安全的编码%23 代替不安全的 ASCII 字符#）将产生图 3-3 中显示的结果。

图 3-3　在浏览器中运行查询

现在，我们可以用 WebRequest 从.NET 运行上述查询，但是，仍然有一个问题需要处理：我们得到的是一个 JSON 文档（其中的条目是问题列表），仍然需要创建一个代表这些条目的.NET 类，然后将每个条目反序列化为该类。虽然这个任务并不特别复杂，但是很乏味。如果有一个机制能够查看该模式，完成为其创建合适类型并将响应加载到这些类型中的枯燥工作就好了。好消息是，这就是类型提供程序所实现的功能。

3.1.2　使用 JSON 类型提供程序

类型提供程序是一种开放机制，也就是说，任何人都可以实现。因此，除了 F# 3.0 开始发货时 Microsoft 发行的几个类型提供程序之外，F#社区已经实现了许多针对不同资源的类型提供程序。大部分都合并到了 FSharp.Data 库中。该库是社区中主流类型提供程序的事实参考，有很好的维护和文档。

NUGET 包和 F#脚本

要在 F#脚本中使用库，必须使用#r 指令引用。举个例子，在脚本中运行如下两行程序，可以使用 libraryAssembly.dll 中的 MyLibrary 库：

```
#r "path-to-the-dll/libraryAssembly.dll"
open MyLibrary
```

使用 NuGet 包的简便方法之一是安装该包，在解决方案资源管理器中的“引用”（Reference）文件夹中找到它，将“属性”（Properties）窗口中的完整路径复制/粘贴到脚本中，

如图 3-4 所示。

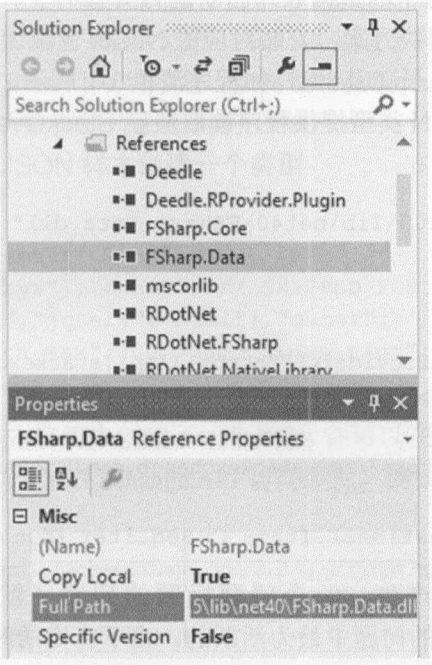


图 3-4 读取程序集完整路径

但是，这种方法有一些问题。默认情况下，NuGet 包下载到解决方案中的“包”（Packages）文件夹，每个包保存在包含其名称和特定版本号的一个文件夹中。因此，使用默认的 Visual Studio 项目结构，你的脚本将是：

```
#r @"\Mathias\Documents\Visual Studio 2013\Projects\ProjectName\packages\
PackageName.1.2.3\lib\net40-full\Assembly.dll"
```

这明显不是很好：路径依赖于本地机器和版本号，两者都可能变化。你可以使用相对路径和#I 指令解决第一个问题，后者添加程序集的搜索路径：

```
#I "..\packages"
#r @"PackageName.1.2.3\lib\net40-full\Assembly.dll"
```

版本号的问题更难解决，每当更新到库的新版本，脚本就会出错，直到手工修复路径为止。.NET 依赖性管理库 Paket（http://fsprojects.github.io/Paket/）为该问题提供了一个简洁的解决方案。我们将在本书余下的部分中使用版本特定路径，因为这种路径的好处是澄清所使用的版本。

在 FSharp.Data 中可以找到的类型提供程序之一是 JSON 类型提供程序。它采用典型的规范类型提供程序的工作方式。涉及的步骤如下：

- 首先，查看样板数据以创建类型。
- 然后，将实际数据源中的数据导入已创建的类型。

我们创建一个新的 F#解决方案，安装对应的 NuGet 包添加对 FSharp.Data 的引用，在默认 Script.fsx 文件中引用 FSharp.Data 库，如程序清单 3-1 所示。

程序清单 3-1　用 JSON 类型提供程序读取最新 StackOverflow C#问题

```
#I @"..\packages\"
#r @"FSharp.Data.2.2.0\lib\net40\FSharp.Data.dll"

open FSharp.Data

type Questions = JsonProvider<"""https://api.stackexchange.com/2.2/questions?
site=stackoverflow""">

let csQuestions = """https://api.stackexchange.com/2.2/questions?site=stackoverflow&
tagged=C%23"""
Questions.Load(csQuestions).Items |> Seq.iter (fun q -> printfn "%s" q.Title)
```

运行这段样板代码将产生 30 个问题标题的列表（默认页面行数为 30），这是 StackOverflow 上标签为 C#的问题中最为活跃的。由于查询的时间相关特性，结果在不同机器上有所不同。

提示：F#支持两种包含特殊字符的字符串处理机制。转义字符串将忽略转义字符（如“\”），允许使用双引号在字符串中嵌入引号，如：printfn @"He said ""hello""!"。三引号也将忽略转义字符，但是允许直接在其中使用引号，如：printfn """He said "hello"!"""。

那么，这里发生了什么呢？第一部分代码 Questions = JsonProvider<...>根据样本数据创建类型 Questions。样本数据通过网络从我们提供的 URL（如前所述，称作 StackExchange API）上访问，以 JSON 文档的形式返回 30 个最新的问题。类型提供程序扫描样本，按照需要创建相应的属性和类型。

例如，如果你在 Visual Studio 中的脚本文件里输入 Questions，等待智能感知功能给出提示，就会有两个发现。首先，Questions 包含几个根据样本内容自动创建的类：Item、Owner 和 Root。在下拉菜单上单击智能感知显示的 Item，应该能够看到输出的一些属性，如 member AcceptedAnswerId:Option<int>。根据样本，TP 已经生成了具有合适类型的类（可接受的答案 ID 如果存在，应该是一个整数，但是如果没有任何满意的答案，ID 也可能不存在）。

Questions 还将输出一个 Load 方法。这个方法可用于从特定来源获取数据，预计返回适合 Questions 类型的某些数据。这就是脚本第二部分所进行的工作：我们传入请求标签为 C#（编码为 C%23）的最新问题的查询，类型提供程序在网上发出查询，接收 JSON 文档，并自动转换为成熟的类型。

JSON 类型提供程序可以稍有不同的方式使用，这有助于帮助澄清此处所用的机制。你应该直接为其提供一个样本（文件或者内嵌字符串），而不是调用 StackExchange API 生成 Questions 类型。程序清单 3-2 说明了这个过程：我们以普通字符串的形式提供从 API 得到的 JSON 输出有效样本（用[<Literal>]属性标记，该属性将其编译为字面常量），并将样本传递给类型提供程序，根据样本的数据结构创建新的 F#类型。

程序清单 3-2 从本地 JSON 样本创建一个类型

```
[<Literal>]
let sample = """{"items":[
{"tags":["java","arrays"],"owner": // SNIPPED FOR BREVITY"},
{"tags":["javascript","jquery","html"],"owner": // SNIPPED FOR BREVITY"},],
"has_more":true,"quota_max":300,"quota_remaining":299}"""
type HardCodedQuestions = JsonProvider<sample>
```

此时，编译器已经根据样本字符串创建了一个类型，你可以在一个预期返回相同结构 JSON 响应的查询中传递该类型，从 StackExchange API 访问数据。例如，获取与 Java 相关的问题：

```
[<Literal>]
let javaQuery = "https://api.stackexchange.com/2.2/questions?site=stackoverflow&tagged=java"

let javaQuestions = HardCodedQuestions.Load(javaQuery)
```

换言之，编译器使用样本生成一个类型。JSON 来自于本地文件还是网络并不重要。一旦创建了类型，就可以使用它加载具有对应形式的数据。例如，如果我们使用这个类型调用不同的 API 端点，就会发生运行异常。

两种方法之间的有趣差异之一是，如果模式随时间变化，它们将有不同的含义。例如，想象一下，Title 属性更名。在第一种情况下，我们将得到编译时的错误，因为每次构建时都会通过访问 API 创建该类型，编译器将发现我们使用的是不存在的属性。相比之下，使用“本地样本”时，我们得到的是运行时的错误：类型基于过时的样本，仍然包含旧属性，这种失配现象只有在访问实际数据时才会捕捉到。

3.1.3 构建查询问题的最小化 DSL

StackExchange API 相当简单明了，随时通过手工编写查询不是很困难。然而，这将在代码中引入许多重复，可能造成打字错误和缺陷。让我们避免这种情况，创建一个最小化的领域特定语言（常常被称作 DSL）。

你可能已经注意到，StackExchange API 请求遵循一种模式：从基本 URL（https://api.stackexchange.com/2.2/）开始，附加一个字符串，编码感兴趣的实体（问题），然后是一个问号和进一步规定查询的可选参数（?site=stackoverflow&tagged=C%23;F%23）。简化查询构造的方法之一是利用该模式，创建一个小型函数描述修改基本查询适应需求的方式，

这样，就可以将上述操作组合成一个精细、容易理解的管道。参见程序清单 3-3。

程序清单 3-3　构建最小化查询 DSL

```
let questionQuery = """https://api.stackexchange.com/2.2/questions?site=stackoverflow"""

let tagged tags query =
    // join the tags in a ; separated string
    let joinedTags = tags |> String.concat ";"
    sprintf "%s&tagged=%s" query joinedTags

let page p query = sprintf "%s&page=%i" query p

let pageSize s query = sprintf "%s&pagesize=%i" query s

let extractQuestions (query:string) = Questions.Load(query).Items
```

本质上，questionQuery 定义作为起点的基本查询。后面的 3 个函数为该查询附加可选的细节，最后一个函数 extractQuestions 尝试使用已构建的查询串加载数据，然后读取对应的条目。注意，我们没有必要提供类型注释，因为 sprintf 包含足够的线索，编译器可以推导出正确的类型。

你的注意力应该集中在一个模式上：每个函数都以查询作为最后一个参数。从 C#看来，这很不寻常。C#方法的参数通常按照重要性排序，从必不可少的参数开始，逐步转向较不重要的参数。在 F#代码中将经常看到这种“相反的”模式，因为这种模式可以使用管道向前操作符|>将函数组成工作流。管道向前操作符可以用前一个函数求得的值作为函数的最后一个参数，例如，add 42 1 等价于 1 |> add 42。

在我们的例子中，可用非常容易理解的代码读取100个标签为C#、100个标签为F#的问题，参见程序清单3-4。

程序清单 3-4　使用我们的 DSL 提取标签为 C#和 F#的问题

```
let ``C#`` = "C%23"
let ``F#`` = "F%23"

let fsSample =
    questionQuery
    |> tagged [``F#``]
    |> pageSize 100
    |> extractQuestions

let csSample =
    questionQuery
    |> tagged [``C#``]
    |> pageSize 100
    |> extractQuestions
```

■ 提示：包含在两个倒引号之内的任何字符串（如“C#”）都是有效的标识符。这对于使代码更易于理解很有帮助。例如，在这种情况下，“C#”的含义比“C%23”更显而易见。这种特征的常见用途是测试：可以简单地使用“The price should be positive”()，而不用将测试方法命名为 The_Price:Should_Be_Positive()。

引用 JSON 类型提供程序和编写一个最小化 DSL 需要 7 行代码。现在，我们已经为数据的改造和探索做好了全部准备。例如，比较与两个组最常关联的标签或者主题只需要几行代码，如程序清单 3-5 所示。

程序清单 3-5 按照语言比较标签

```
let analyzeTags (qs:Questions.Item seq) =
    qs
    |> Seq.collect (fun question -> question.Tags)
    |> Seq.countBy id
    |> Seq.filter (fun (_,count) -> count > 2)
    |> Seq.sortBy (fun (_,count) -> -count)
    |> Seq.iter (fun (tag,count) -> printfn "%s,%i" tag count)

analyzeTags fsSample
analyzeTags csSample
```

以 Item 的一个序列作为输入，我们收集标签，将由多个序列组成的序列展开为单一序列，计数每个标签，过滤出现次数少于两次的标签，按照计数降序排列，最后打印输出标签和计数。

■ 提示：“解包”元组时，下划线用作通配符。这省去了为未用变量创建一个名称的麻烦，还有助于通知你正在使用的是元组中的哪些元素。

如果运行上述代码，可能会观察到稍有不同的结果，因为查询是与时间相关的。但是，我猜想结果和我观察到的类似（见下表）。

表 最常与 C#和 F#相关的标签

C#标签	F#标签
c#,100	f#,100
wpf,13	f#-data,10
.net,13	type-providers,7
asp.net,11	functional-programming,6
entity-framework,5	f#-3.0,6
asynchronous,5	c#,5
asp.net-mvc-4,5	.net,5
sql,5	f#-interactive,5
linq,4	
xaml,4	
windows-store-apps,4	

人们感兴趣的 C#主题中，大部分围绕客户-服务器技术和 Web，而 F#更多地以数据为中心（到此刻应该不会让你觉得太过意外）。有趣的是，C#问题所相关的技术中大部分都可以用 F#类型提供程序使其大大简化（sql 和数据访问，xaml 等）。

在本章中，我们的目标不是全面分析 StackOverflow（第 5 章将研究这个问题），而是说明类型提供程序和 F#可以使各种来源的数据处理变得极其简单。我们以 JSON 的例子作为焦点，但是大部分面向数据的类型提供程序（CSV、XML 等）遵循类似的模式——首先获取数据样本以推导需要创建的类型，然后使用创建的类型加载数据。本节对 StackOverflow 的探索告一段落——你可以随意进行更多的探索！

3.2　世界上的所有数据

我们已经了解了类型提供程序的定义，以及它们对数据编程的显著简化作用，下面我想聚焦于与数据科学家关系特别紧密的两个主题。第一个是 Deedle，这个.NET 数据框架库显著地简化了数据集的操纵与分析。第二个是 R 类型提供程序，它使 R 统计语言可从 F#中使用，并揭示了巨大的统计及可视化软件包生态系统。

3.2.1　世界银行类型提供程序

世界银行（World Bank）是一个联合国国际金融机构，为发展中国家资本项目提供贷款。在维护和向公众提供极其丰富的社会经济学信息数据集方面也很著名。你可以在 http://data.worldbank.org/中了解这些数据集的内容。世界上 200 多个国家中的人们都可以下载数十年间 1400 多个指标的年度数据，涵盖了从经济与发展到性别或者基础设施等方面。

那么，接下来的事情就都是小儿科了吗？并非如此。问题是，虽然数据就在那里，获取工作却有些乏味。比如，如果我想获取世界上每个国家 2000 年和 2010 年的人口数据以及当前的陆地面积，就必须进行许多搜索工作，以选择对应的国家、指标和年份。我所获得的是一个文件（Excel、CSV 或者 XML），可能仍然必须进行一些处理，才能使其成为适合需求的形式。而且，如果我不知道名称，找到所关心的数据序列（或者国家）并不总是轻而易举的。

注意，这绝不是在批评世界银行的网站——它实际上非常优秀。问题是，创建一个用户界面以自由选择用户所需的数据，同时在巨大的数据库中以层次化方式导航，是很困难的。这时最方便的是有一种不同的机制，使我可以随时查询和发现数据——这正是 FSharp.Data World Bank 类型提供程序的功能。它遵循的模式和我们迄今为止所看到的略有不同。它不使用模式信息为通用数据格式创建类型，而是专为数据源编写，唯一的目的就是在花费精力最小的情况下发现和使用数据。

使用 World Bank 类型提供程序需要安装 FSharp.Data NuGet 包，在脚本中添加对该包的引用，创建一个数据上下文，上下文将建立与世界银行网站的连接，如程序清单 3-6 所示。

程序清单 3-6 使用 World Bank 类型提供程序

```
#I @"..\packages\"
#r @" FSharp.Data.2.2.0\lib\net40\FSharp.Data.dll"

open FSharp.Data

let wb = WorldBankData.GetDataContext ()
wb.Countries.Japan.CapitalCity
```

一旦创建了数据提供程序，就可以使用它直接访问世界银行网站，用智能感知功能发现可用数据（参见图 3-5）。

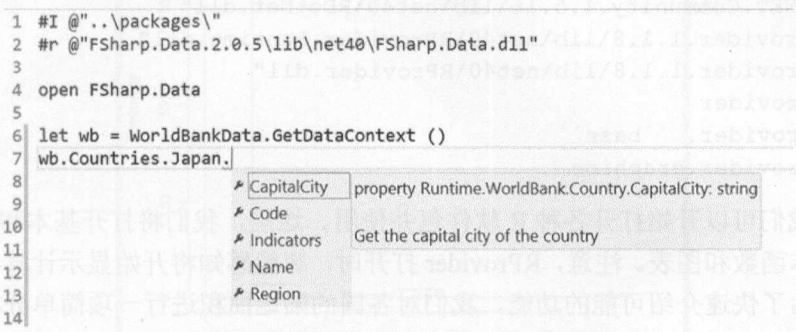


图 3-5 用智能感知功能浏览世界银行数据

在此，我们要做的就是读取 2000 年和 2010 年世界各国的人口数据，这相当简单：

```
let countries = wb.Countries

let pop2000 = [ for c in countries -> c.Indicators.``Population, total``.[2000]]
let pop2010 = [ for c in countries -> c.Indicators.``Population, total``.[2010]]
```

World Bank 类型提供程序为我们提供了宝贵数据的直接访问功能。话虽如此，现在我们又碰到了一个新问题，如何从中提取有趣的信息？浏览数百个国家的数据点，希望注意到其中的模式不是挖掘数据以发现有趣事实的最有效手段。应该有一些统计工具供我们使用——幸运的是，我们已经有了这种工具，这要感谢 R 类型提供程序。

3.2.2 R 类型提供程序

R 是“用于统计计算和图表的免费软件环境”，按照该组织（www.r-project.org）的说法，这是统计学家为统计学家设计与实现的一种编程语言。R 语言是开放源码的，完全免费，并且自带了 R 社区创建的令人惊叹的软件包生态系统，覆盖了统计学的所有领域。

R 类型提供程序是一个有趣的创造。目前，我们已经看到了聚焦于数据可发现性的例子。R 提供程序完成的工作与此类似，但是适用于完全不同的外部资源类型：一种编程语言。假

定你已经在机器上安装了 R 语言，R 提供程序将发现已经安装的软件包，然后让你在 F#环境中打开它们，就像它们是普通的.NET 库一样。R 提供程序还能够发现可用的 R 函数，并从 F#中直接调用。

安装 R：要想使用 R 类型提供程序，需要在机器上安装 R。R 是完全免费的开放源码语言，可以运行在所有平台上。你可以在 http://www.r-project.org/上找到关于 R 的指南和文档。注意，虽然 R 在所有平台上都可以运行得很好，但是在本书编写的时候，R 类型提供程序对 mono 只有部分支持。

让我们来看看 R 提供程序能做什么。首先，安装 NuGet 包 RProvider，并在脚本中添加必要的引用：

```
#r @"R.NET.Community.1.5.16\lib\net40\RDotNet.dll"
#r @"RProvider.1.1.8\lib\net40\RProvider.Runtime.dll"
#r @"RProvider.1.1.8\lib\net40\RProvider.dll"
open RProvider
open RProvider. ``base``
open RProvider.graphics
```

此时，我们可以开始打开各种 R 软件包并使用。这里，我们将打开基本和图表软件包，它们输出基本函数和图表。注意，RProvider 打开时，智能感知将开始显示计算机上实际存在的软件包。为了快速介绍可能的功能，我们对各国的陆地面积进行一项简单分析（参见程序清单 3-7）。

程序清单 3-7　R 语言基本摘要统计

```
// Retrieve an (F#) list of country surfaces
let surface = [ for c in countries -> c.Indicators.``Surface area (sq. km)``.[2010]]
// Produce summary statistics
R.summary(surface) |> R.print
```

我们用 World Bank 类型提供程序读取每个国家的陆地面积，将数值保存在一个 F#列表中。使用 R（再次通过智能感知）将显示环境中当前可用的函数。首先，调用 R.summary 产生表面积变量的一个摘要，然后调用 R.print 生成如下的“优质打印结果”：

```
Min.    1st Qu.     Median    Mean   3rd Qu.      Max.    NA's
   2      11300      99900  630200   450300  17100000       1
```

注意：如果你只运行 R.summary，可能会注意到它返回一个 RDotNet.SymbolicExpression。为了避免 R 和.NET 类型之间不必要且代价很高的转换，R 类型提供程序一般会通过 R 符号表达式（Symbolic Expression，R 数据和运算的表现形式）向 R 语言发送信息，并尽可能保持这种形式，只在请求时转换回.NET 类型。

这里没有发生什么惊天动地的大事，但是上述功能使用非常方便。摘要中返回最小值和最大值、平均值、3 个四分位值（大于 25%、50%和 75%样本的值）以及漏失值（NA）

的个数。我们一眼就可以看出陆地面积的差别很大，从最小值 2 平方公里到最大值 1700 万平方公里——有一个国家没有数据。这些摘要很有用，但是为了说明数值的分布情况，图表有助于更好地了解数据的“样子”。用 R.hist(surface)函数很容易做到这一点，该函数产生图 3-6 中的图表。

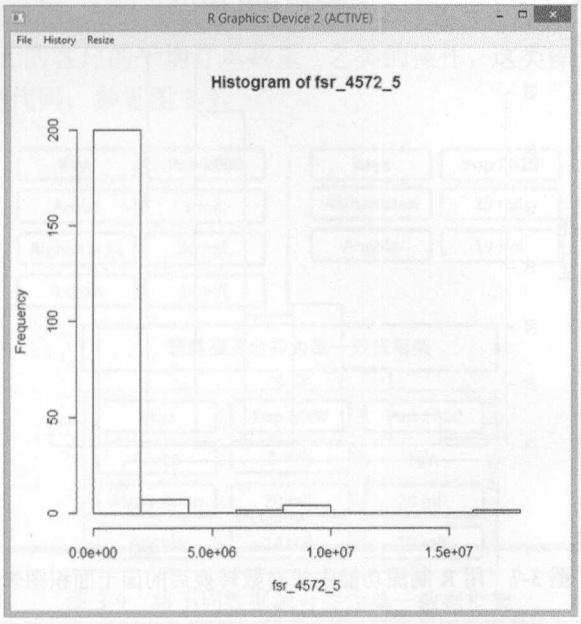


图 3-6 R 制图功能生成的国家面积简单直方图

这个图表不是很好，X 轴显示的变量使用了难以理解的名称 fsr_4572_5，而没有使用“面积”。我们将在后面看到如何创建更漂亮的图表，但是请记住，生成这个图表只需要 R.hist 的一次调用，这对于探索工作来说非常完美，很快就显示了我们需要知道的信息——面积的分布很不均匀，相对小的国家占大多数，只有少数“巨无霸”。

如果我们想更好地了解中等水平国家的情况，可以绘制国土面积的对数图表。最好的一点是，我们可以使用自己认为最适合于工作的任何工具：F#代码或者 R 函数。例如，现在我们可以使用 R.hist(surface |> List.map log)这样的 F#代码。然而，R.log 函数更简洁，所以我们使用 R.hist(surface |> R.log)，生成图 3-7 所示的图表。

直方图是理解特征分布的好手段。但是，数据可视化的主力是散点图——在二维图表上“投射”数据点。这是探索特征之间关系的方便手段，例如，我们可能不知道某个国家的面积与人口之间的关系。回答这一问题的方法之一是使用 R.plot(surface, pop2010)绘制图表，在一个轴上表现人口数量，在另一个轴上表现国土面积，并观察结果，如图 3-8 所示。

从图形看来，面积和人口似乎只有有限的关系，散点图上没有明显的模式。

这一简单的场景说明，R 可以生成简单的统计数字和图表，帮助我们更好地理解单独的特征。然而，如果数据集较大，每次研究一个特征很快就会成为梦魇。通过数据进行挖掘时，

一般会得到许多特征，必须快速探索它们的关系，合并现有特征、将其重组为新的特征。这也是我们接下来所要探索的主题（数据框架）成为数据科学家手中流行工具的原因。

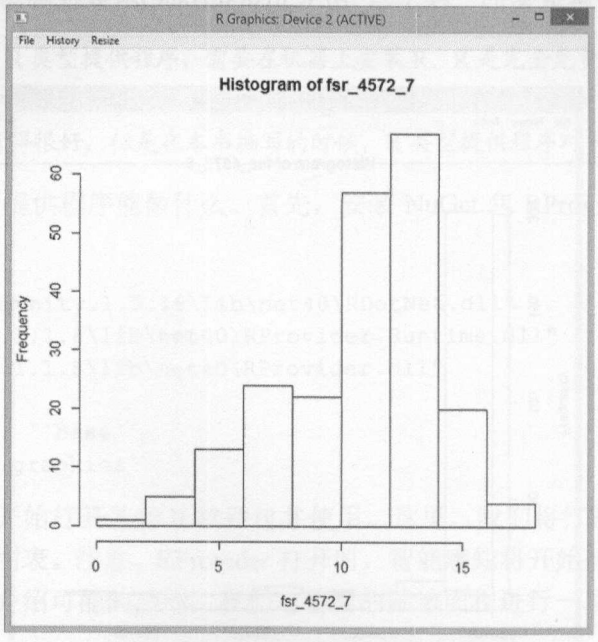


图 3-7　用 R 制图功能生成对数转换后的国土面积图表

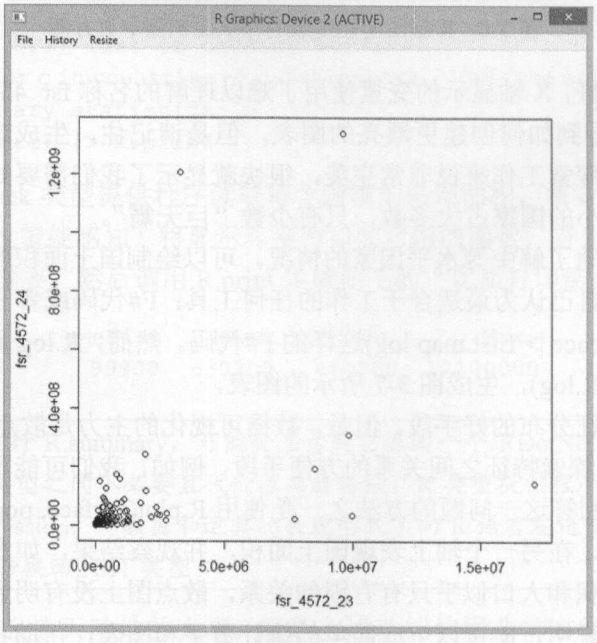


图 3-8　使用 R.plot 绘制基本散点图

3.2.3 分析数据与 R 数据框架

简单地说，数据框架是一种以观测值集合操纵为目的的数据结构。每个观测值对应于数据框架中的一行，也可以按列访问单独特征。在这个意义上，数据框架和数据库表很类似，但是面向的是内存中的操纵。这本身就已经很方便：单独的行上保留类型信息，但是简化了“计算收入高于 100 万元的客户的平均订单数量”之类的操作，这类操作并不难，但是需要一次又一次地重复类似的代码。参见图 3-9。

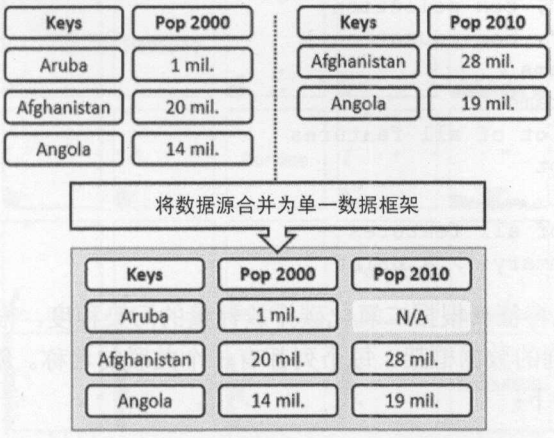


图 3-9 将不同数据源合并为单一数据框架

数据框架的另一个主要目的是协助将数据源合并为单一框架。数据框架本身通常依赖键码识别每行数据，这样不同特征可以通过正确的行数关联，然后连接。另一个重要的相关功能是漏失值的管理：如果我试图关联两个特征——比如国家的面积和人口——其中某些特征的数据不完整，我就会希望结果数据框架包含所有国家的并集，但是标明哪个国家有漏失数据，并在计算某些统计值（如平均面积）时不会造成太大问题的情况下处理漏失值。最后，数据框架通常支持动态列创建。在前一个例子中，我可能想要观察新特征，如国家的人口密度（人口数量除以国土面积）；数据框架通常允许你添加一个新列，以简单的方式表达列组合。

数据框架是 R 的一个核心类型。今天，数据科学家所使用的大部分编程语言都有数据框架库。它们通常是受到 R 数据框架的启发，但是根据语言特征而采用不同的风格。.NET 也没有例外：有几个好的商业化产品和一个出色的开源库 Deddle，我们将在本书中做简要的阐述。

在介绍 Deedle 之前，我们先花一点时间了解如何使用原生 R 数据框架。R 类型提供程序自带几个实用函数，使从 F#中创建 R 数据框架更加容易。作为一个例子，尝试程序清单 3-8 中的代码。

程序清单 3-8　创建 R 数据框架并绘制对应图表

```
let pollution = [ for c in countries -> c.Indicators. ``CO2 emissions (kt) ``.[2000]]
let education = [ for c in countries -> c.Indicators. ``School enrollment, secondary
(% gross) ``.[2000]]

let rdf =
    ["Pop2000", box pop2000
     "Pop2010", box pop2010
     "Surface", box surface
     "Pollution", box pollution
     "Education", box education ]
    |> namedParams
    |> R.data_frame
    // Scatterplot of all features
    rdf |> R.plot

    // Summary of all features
    rdf |> R.summary |> R.print
```

我们读取两个其他特征（根据二氧化碳排放计量的污染程度，根据中学入学率计量的教育水平），创建包含 5 列的数据框架，每个列都有一个关联的名称。创建数据框架之后，可以生成一个摘要，输出如下：

```
>
    Pop2000              Pop2010             Surface          Pollution
Min.   :9.419e+03   Min.   :9.827e+03   Min.   :     2   Min.   :   15
1st Qu.:6.255e+05   1st Qu.:7.981e+05   1st Qu.: 11300   1st Qu.: 1050
Median :5.139e+06   Median :5.931e+06   Median : 99900   Median : 6659
Mean   :2.841e+07   Mean   :3.207e+07   Mean   : 630216   Mean   : 122608
3rd Qu.:1.631e+07   3rd Qu.:2.097e+07   3rd Qu.: 450300   3rd Qu.: 53601
Max.   :1.263e+09   Max.   :1.338e+09   Max.   :17098240   Max.   :5713560
NA's   :1 NA's                                          :20
   Education
Min.   : 6.051
1st Qu.: 41.699
Median : 78.588
Mean   : 70.782
3rd Qu.: 93.882
Max.   :160.619
NA's   :70
```

我们甚至可以将整个数据框架传递给 R.plot，同时可视化所有特征，如图 3-10 所示。

此类图表非常实用，一眼看去，Pop2000 和 Pop2010 之间的相关性显而易见：它们的散点图几乎成为一条直线。Education（教育）似乎独立于其他 4 个特征（没有出现明显的视觉特征），Surface Area（国土面积）和 Pollution（污染）似乎和 Population（人口）有某种关联

（人口越多，前两个值可能越高）。还要注意，虽然我们有许多漏失值，但是 R 并未因此摇摆不定，而是优雅地处理了这一问题。

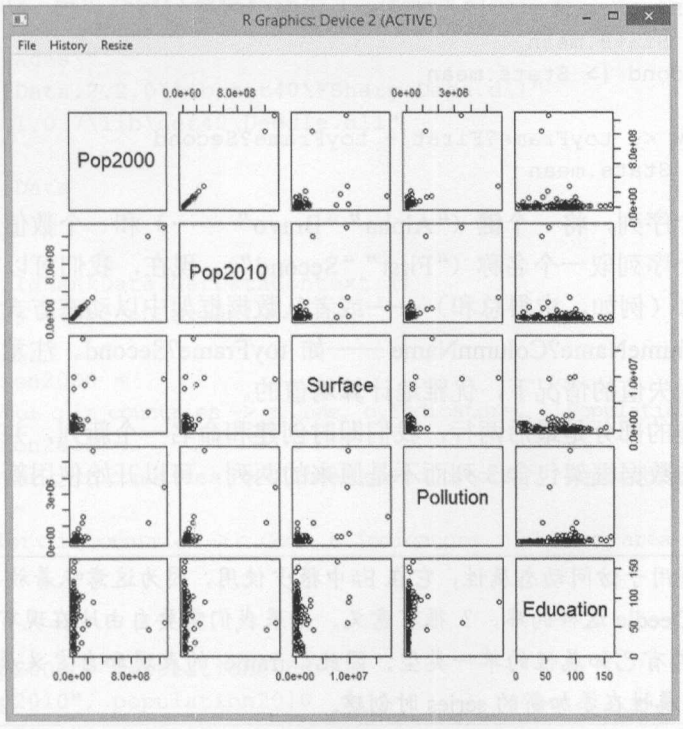


图 3-10 可视化特征之间的关系

在此，细节并不特别重要，我希望指出的要点是，在多个特征的处理中，数据框架很方便。但是，除了本例这样的简单情况之外，在 F#中使用 R 数据框架操纵数据相当令人不快。这是 Deedle 的切入点：它原生于.NET，语法设计是与.NET 友好的（对 C#和 F#都是如此），但是可以使用 Deedle RPovider 插件直接向 R 提供数据。

3.2.4 .NET 数据框架 Deedle

Deedle 的两个核心组成部分是序列和框架。**序列**（Series）是一个键-值对集合。序列可以合并为一个数据框架并根据键匹配，自动管理漏失值。我们首先从一个“玩具”例子开始说明 Deedle 的工作方式，然后将其用于更有趣的场景（参见程序清单 3-9）。

程序清单 3-9 用 Deedle 创建序列和数据框架

```
#r @"Deedle.1.0.7\lib\net40\Deedle.dll"
open Deedle

let series1 = series [ "Alpha", 1.; "Bravo", 2.; "Delta", 4. ]
```

```
let series2 = series [ "Bravo", 20.; "Charlie", 30.; "Delta", 40. ]
let toyFrame = frame [ "First", series1; "Second", series2 ]

series1 |> Stats.sum
toyFrame |> Stats.mean
toyFrame?Second |> Stats.mean

toyFrame?New <- toyFrame?First + toyFrame?Second
toyFrame |> Stats.mean
```

我们创建两个序列，将一个键（“Alpha”“Bravo”……）和一个数值关联，并从中创建一个框架，为每个序列取一个名称（“First”“Second”）。现在，我们可以在序列或者整个数据框架上执行计算（例如，求得总和）——或者从数据框架中以动态方式访问每个序列，使用的语法为 DataframeName?ColumnName ——如 toyFrame?Second。注意数据框架是如何在两个序列中都有漏失值的情况下，优雅地计算均值的。

上例中最有趣的部分是最后两行，我们即时创建和命名一个新列，并根据其他列做出定义。现在，我们的数据框架包含 3 列而不是原来的两列，可以开始使用新列，就像它从一开始就存在似的。

提示：?操作符用于访问动态属性，它在 F#中很少使用，因为这意味着放弃静态类型的安全性。然而，对于 Deedle 这样的库，？很有意义。如果我们需要自由地在现有框架中添加数据，就不能提前创建具有已知属性的单一类型。因此，frame 的表现和自定义属性对象（Expando Object）类似，新属性在添加新的 series 时创建。

Deedle 还有许多这里没有展现的特性，我鼓励你前往文档页面（http://bluemountaincapital.github.io/Deedle/）自行了解 Deedle 的功能。这个例子的意图是让你对数据框架有所认识，提供 Deedle 的某些关键思路，以帮助入门。下面，我们来看一个 R 和 Deedle 集成的例子。

3.2.4　全世界的数据统一起来！

在前一小节中，我们开始观察世界人口数量，从世界银行类型提供程序中获取数据。超过某一个程度之后，就难以在大数据集中辨别任何模式，对于超过 200 个国家和地区的列表，我们需要其他工具。地图是真正有效的工具。幸运的是，在现有的许多 R 软件包中，有一个是 rworldmap，它的功能正如其名：创建世界地图。方法之一是简单地将数据与国家或地区关联，然后让该软件包完成自己的魔法。

我们从绘制 2000 年和 2010 年各国人口的图表入手。首先，创建一个 Deedle 数据框架，并用来自世界银行的数据填充。为此，我们需要构建一个序列 ——如果你还记得，这就是键-值对的集合。我们希望该信息可以供 rworldmap 使用，以便将正确的数据点关联到对应的国家或地区。事实证明，rworldmap 和世界银行类型提供程序都支持 ISO3 编码 ——唯一标识国家或地区的 3 字符代码，我们用它作为键码。

为了清晰起见，我们创建一个新的脚本文件 WorldMap.fsx 并构建数据框架，如程序清单 3-10 所示。

程序清单 3-10 用世界银行数据构建一个 Deedle 数据框架

```
#I @"..\packages\"
#r @"FSharp.Data.2.2.0\lib\net40\FSharp.Data.dll"
#r @"Deedle.1.0.7\lib\net40\Deedle.dll"

open FSharp.Data
open Deedle

let wb = WorldBankData.GetDataContext ()
let countries = wb.Countries

let population2000 =
    series [ for c in countries -> c.Code, c.Indicators. ``Population, total``.[2000]]
let population2010 =
    series [ for c in countries -> c.Code, c.Indicators. ``Population, total``.[2010]]
let surface =
    series [ for c in countries -> c.Code, c.Indicators. ``Surface area (sq. km)``.[2010]]

let ddf =
    frame [
        "Pop2000", population2000
        "Pop2010", population2010
        "Surface", surface ]
ddf?Code <- ddf.RowKeys

#r @"R.NET.Community.1.5.16\lib\net40\RDotNet.dll"
#r @"RProvider.1.1.8\lib\net40\RProvider.Runtime.dll"
#r @"RProvider.1.1.8\lib\net40\RProvider.dll"
#r @"Deedle.RPlugin.1.0.7\lib\net40\Deedle.RProvider.Plugin.dll"let dataframe =
    frame [
        "Pop2000", population2000
        "Pop2010", population2010
        "Surface", surface ]
dataframe?Code <- dataframe.RowKeys
```

我们简单地读取 2000 年和 2010 年的人口数量，以及国土面积，创建 3 个序列，以每个国家或地区的国家编码作为键。然后，创建一个 Deedle 数据框架，为每个序列取一个唯一的名称，最后加入另一个序列“Code”，直接输出国家或地区编码（我们需要这个序列，以便将地图中的标题与国家或地区数据绑定）。

现在数据框架中已经有了数据，我们必须创建一个地图，定义每个国家或地区与数据的关联方式。这相当简单，如程序清单 3-11 所示。

程序清单 3-11　用 rworldmap 创建一个地图

```
#r @"R.NET.Community.1.5.16\lib\net40\RDotNet.dll"
#r @"RProvider.1.1.8\lib\net40\RProvider.Runtime.dll"
#r @"RProvider.1.1.8\lib\net40\RProvider.dll"
#r @"Deedle.RPlugin.1.0.7\lib\net40\Deedle.RProvider.Plugin.dll"

open RProvider
open RProvider.``base``
open Deedle.RPlugin
open RProvider.rworldmap

let map = R.joinCountryData2Map(dataframe,"ISO3","Code")
R.mapCountryData(map,"Pop2000")
```

我们打开 Deedle R 插件（它使 Deedle 框架，可直接用于 R）和 rworldbank 包，然后创建一个地图，定义数据来源（dataframe ddf），使用“Code”列（映射到世界银行国家或地区编码）作为 ISO3 编码（rworldbank 的内建选项之一）定义地图上的每个国家或地区与数据框架中数据的映射。这样就准备好了：现在运行 R.mapCountryData(map,"Pop2000")，将从数据框架中读出 Pop2000 列的所有值，匹配 Code 列和 ISO3 国家或地区编码，将其与国家或地区相关，生成图 3-11 所示的地图。对于 10 多行代码来说，这结果还不错！

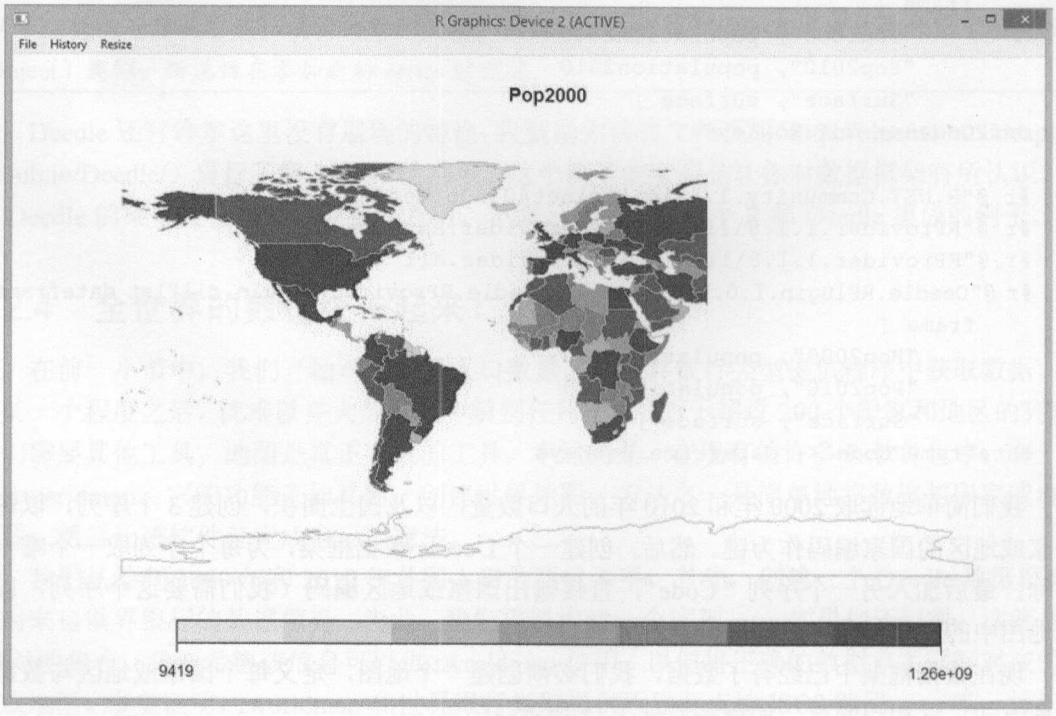


图 3-11　2000 年世界人口地图

■ **注意**：安装 R 软件包。当你在机器上安装 R 时，将安装软件包的最小集合，该集合提供基本功能。但是，R 的关键优势之一是巨大的软件包生态系统，该系统为所有可视化或者统计场景提供了解决方案。R 自带一个内建包管理器，可以轻松地在本地机器上下载和安装软件包。例如，这里使用的 rworldmap 包不是标准软件包。要安装它，可以从 R 环境中调用 install.packages("rworldmap") 或者从 F# Interactive 中使用 open RProvider.utils 和 R.install_packages(["rworldmap"])，利用 R 类型提供程序。这将提示一个下载站点，然后在机器上安装软件包和依赖模块，随后就可以使用软件包了。

这是一个漂亮的地图，也是在数据集中快速找出模式的更好方法。然而，对我们的启发并不大：在图上可以看到格陵兰岛、喜玛拉雅山脉和撒哈拉沙漠中的人口很少。让我们来看看是否能够生成更有趣的地图。使用 Deedle 等数据框架的好处之一不是操纵原始数据，而是更容易创建新特征。

举个例子，考虑一下国家或地区的人口密度：也就是每平方公里（或者英里）居住的人数。Deedle 轻而易举就可以做到，如程序清单 3-12 和图 3-12 所示。

程序清单 3-12　生成人口密度地图

```
dataframe?Density <- dataframe?Pop2010 / dataframe?Surface
let map = R.joinCountryData2Map(dataframe,"ISO3","Code")
R.mapCountryData(map,"Density")
```

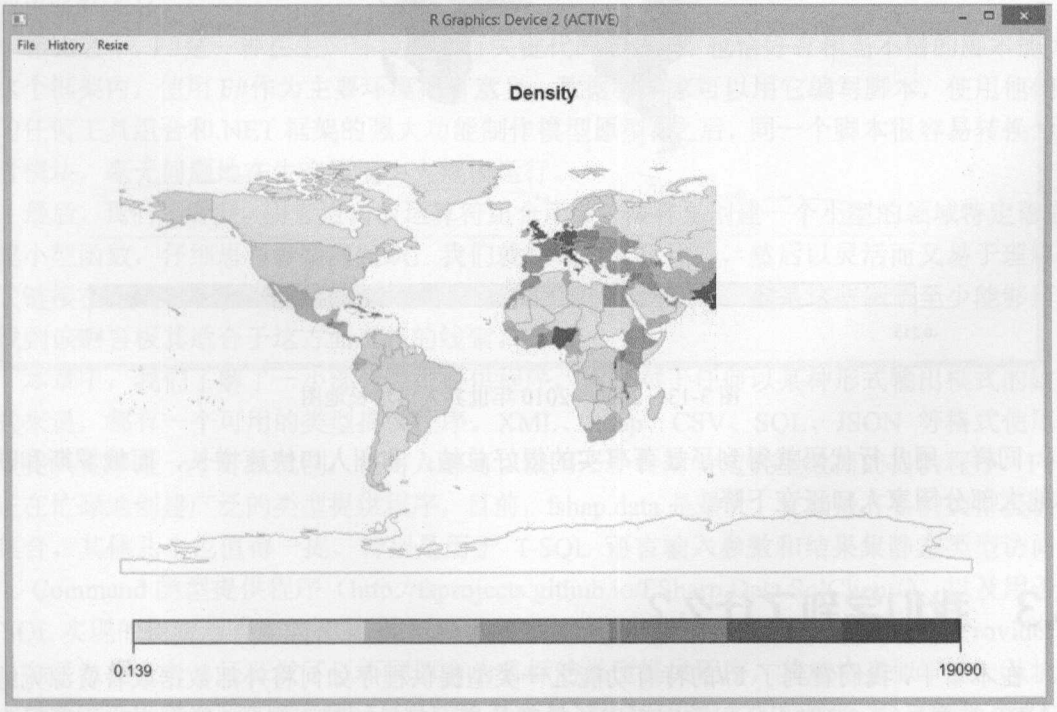


图 3-12　世界人口密度地图

我们即时创建一个名为 Density 的新列，定义为两个其他列的函数，简单地重新将地图绑定到更新的数据框架上。这个地图更加有趣，强调了世界上几个人口稠密区，这些区域位于欧洲、东南亚和非洲。

类似地，我们可以用几行代码可视化世界人口增长的地区，如程序清单 3-13 和图 3-13 所示。

程序清单 3-13　生成人口增长地图

```
dataframe?Growth <- (dataframe?Pop2010 - dataframe?Pop2000) / dataframe?Pop2000
let map = R.joinCountryData2Map(dataframe,"ISO3","Code")
R.mapCountryData(map,"Growth")
```

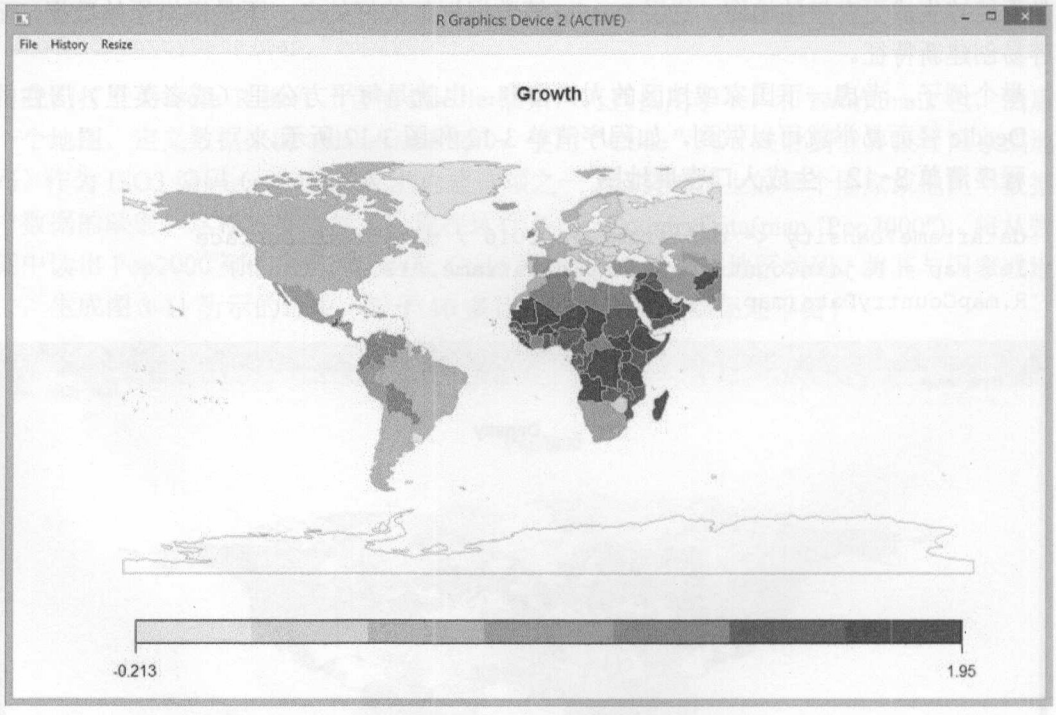


图 3-13　2000—2010 年世界人口增长地图

同样，用几行代码就得到了显著事实的很好总结：非洲人口快速增长，而俄罗斯和欧洲其他大部分国家人口正在下降。

3.3　我们学到了什么?

在本章中，我们看到了 F#的特有功能之一类型提供程序如何将外部数据或者资源无缝引入.NET 类型系统。这对于任何数据探索场景都是显而易见的效率提升。类型提供程序管理一项引人注目的功能：以和动态语言完全相同的方式获取数据，但是使用静态类型数据，为用

户提供来自编译器和工具的额外帮助，避免运行时才能发现的愚蠢错误（如打字错误或者不正确的类型转换）。

更有趣的可能是，有类型的数据还为数据可发现性提供了巨大好处。数据保存格式通常主要是为了解决计算机 I/O 问题，而不是为了易理解性。因此，掌握一个数据集往往是令人不快的经历，至少对人来说是这样。类型提供程序提供了一种非常有趣的机制，用已经使用了数十年的工具帮助开发人员浏览和理解大代码库（复杂的数据集）的内部组织，通过智能感知将其应用到“常规”数据。这是惊人的强大机制。按照我的经验，输入“.”就能发现数据块之间的联系，或者让计算机自动找出与输入部分匹配的项目，是深入数据集、理解其组织方式极有效的手段。

本章的另一个要点是如何使用同一机制，在 F#环境中使用所有 R 资源。R 既有粉丝，也有憎恶者，但是其生态系统的深度不容置疑。如果你有统计学方面的问题，很有可能找到一个软件包。R 实际上具有解决所有统计学问题的软件包，而且以生成漂亮的出版级图表和可视化功能而著称，这些功能都可以在 F#中使用。

R 类型提供程序是 F#作为有效的“胶水”语言的出色范例。使用 RStudio 等 R 专用工具所提供的 R 使用体验比通过类型提供程序从 F#中调用稍好一些。然而，选择 R 作为主要开发环境有很大的意义。R 从未考虑集成，在生产环境中运行或者与其他系统通信也不是它的长处。在此基础上，R 没有任何与类型提供程序接近的功能，从所有地方探索和加载数据的能力也荡然无存。

相比之下，F#是一种在生产环境中运行关键代码的语言，也恰好有相当不错的脚本能力。在这个框架内，使用 F#作为主要环境很有意义。数据科学家可以用它编写脚本，使用他们喜欢的任何工具组合和.NET 框架的强大功能制作模型原型；之后，同一个脚本很容易转换为类或者模块，毫无问题地在生产环境中大规模运行。

最后，我们还看到，用管道向前运算符组合函数，很容易创建一个小型的领域特定语言。创建小型函数，仔细思考参数的组织，我们就能编写几行代码，然后以灵活而又易于理解的方式链接在一起。F#中 DSL 的主题远远多于本章中的简化例子，但是这些例子至少能够帮助你找到该语言极其适合于这方面工作的线索。

本章中，我们了解了一小部分类型提供程序，但是对于任何以某种形式输出模式的数据格式来说，都有一个可用的类型提供程序。XML、Soap、CSV、SQL、JSON 等格式使用的类型提供程序是一个开放机制：任何人都可以编写一个针对特定类型数据的提供程序，F#社区正在忙碌地创建广泛的类型提供程序。目前，fshap.data 是最大、文档最齐备的类型提供程序集合。其他几个也值得一提，特别是用于 T-SQL 语言输入参数和结果集静态类型访问的 SQL Command 类型提供程序（http://fsprojects.github.io/FSharp.Data.SqlClient/），以及用于不同 SQL 实现的探索与查询的 SQL 类型提供程序（https://github.com/fsprojects/SQLProvider）。

除了数据之外，还开发了以不同语言为目标的类型提供程序，其风格类似于 R。从机器学习的角度看，最有趣的是创建 Matlab 和 Python 类型提供程序的持续努力。但是，到本书编写的时候，这些类型的提供程序还没有达到与 R 类型提供程序相同的成熟度。

实用链接

- FSharp.Data 是一组维护和文档都很好的开源 F#类型提供程序：http://fsharp.github.io/FSharp.Data/。
- 世界银行数据网站：http://data.worldbank.org/。
- R 统计环境：http://www.r-project.org/。
- F# R 提供程序：http://bluemountaincapital.github.io/FSharpRProvider/。
- .NET 数据框架库 Deedle：http://bluemountaincapital.github.io/Deedle/。
- Rworldmap 软件包的出色入门教程：http://cran.r-project.org/web/packages/rworldmap/vignettes/rworldmap.pdf。
- .NET 依赖性管理器 Paket：http://fsprojects.github.io/Paket/。
- Fsharp.Data.SqlClient（http://fsprojects.github.io/FSharp.Data.SqlClient）和 SQLProvider（http://fsprojects.github.io/SQLProvider）是专门用于处理 SQL 数据库的两个补充类型提供程序。

第 4 章

自行车与人

运用梯度下降技术，用回归模型拟合数据

迄今为止，我们所处理的问题涉及的都是一组有限类别之间的分类。虽然分类可以应用到许多实用场景下，但是更常见的问题是预测一个数字。例如，考虑如下的任务：给定二手车的特性（车龄、里程数、发动机规格等），如何预测它的售价？这个问题并不适合分类的模式。我们需要的模型至少在两个方面上不同于分类模型：

- 模型预测的是真实的数字，取值范围可能很广，而且可以进行有意义的比较。
- 我们感兴趣的是特征与预测值之间的函数关系：也就是说，特征的变化如何影响预测。

在我们的例子中，我们找这样一个模型：

$$price = \theta_0\theta_1 \times age + \theta_2 \times miles$$

和前面看到的分类模型不同，预测价格不局限于固定的可选项集合，可能取很广泛的值。而且，如果在模型中“插入”不同的里程数值，可以直接观察到特征变化对预测值的影响，可将其与车龄变化相比，以理解价格对不同特征的敏感度。

这类模型称作**回归模型**，它的历史相对较短，重要的发展始于 19 世纪。回归模型起源于物理学，以及根据实验观测值验证物理模型的需求，这奠定了现代统计学的基础。这些方法传播到生物学、经济学和其他领域，不断演变改善，以处理各个学科的特殊问题，现在又成为了机器学习的重要部分。

在本章中，我们的目标是预测指定日期、信息（如天气条件或星期几）条件下使用自行车共享服务的人数。我们将从简单的模型开始，逐步精炼以改进精度，引入重要的机器学习技术。在工作推进过程中，我们将：

- 解释如何将“找出与很好地拟合数据的直线”描述为“找出总体误差最小的直线”。
- 介绍“梯度下降”技术，这种强大的通用技术可以找出函数的最小值。然后，我们使用这种技术创建第一个预测模型。
- 探索线性代数如何帮助编写更简短的模型和算法，加速计算。

- 介绍一种轻松创建和修改模型的方法，使用函数定义选择数据的某一部分作为预测使用的特征。
- 进一步改进我们的预测。说明如何通过处理不同数据类型和创建更复杂的非线性特征，利用更多数据。

4.1　了解数据

在本章中，我们将使用加州大学欧文分校机器学习知识库中的另一个数据集——“自行车共享数据集”，可以在这里找到：https://archive.ics.uci.edu/ml/datasets/Bike+Sharing+Dataset。

该数据集基于“Capital Bikeshare”（首都自行车共享）项目，按照该项目的说明：

Capital Bikeshare 将 2500 多辆自行车放在你的指尖。你可以选择华盛顿特区、弗吉尼亚州阿灵顿和亚历山大、马里兰州蒙哥马利县的 300 多个站点乘车，然后将其还回靠近目的地的任何站点。你可以借用一辆自行车上班、赶地铁、取回干洗的衣服、购物或者拜访朋友及家人。

来源：http://www.capitalbikeshare.com

完整的数据集由 Capital Bikeshare 和其他公共数据来源（如天气和公共假日）共同提供。

4.1.1　数据集有哪些内容？

在深入模型构建部分之前，我们先简单地看看所要处理的数据。数据集本身可以在 https://archive.ics.uci.edu/ml/machine-learning-databases/00275/上找到。Bike-Sharing-Dataset.zip 中包含 3 个文件：day.csv、hour.csv 和 readme.txt。我们将使用的文件是 day.csv，该文件中包含按日汇聚的数据；hour.csv 文件中包含相同的数据，但是按照小时汇聚。Readme.txt 文件中包含的内容留给读者随意猜测！

和往常一样，我们从创建包含一个脚本文件的 F#库项目开始，然后将数据文件 day.csv 添加到项目中。如果在 Visual Studio 中单击该文件，就可以看到数据内容，如图 4-1 所示。

```
1  instant,dteday,season,yr,mnth,holiday,weekday,workingday,weathersit,temp,atemp,hum,windspeed,casual,registered,cnt
2  1,2011-01-01,1,0,1,0,6,0,2,0.344167,0.363625,0.805833,0.160446,331,654,985
3  2,2011-01-02,1,0,1,0,0,0,2,0.363478,0.353739,0.696087,0.248539,131,670,801
4  3,2011-01-03,1,0,1,0,1,1,1,0.196364,0.189405,0.437273,0.248309,120,1229,1349
5  4,2011-01-04,1,0,1,0,2,1,1,0.2,0.212122,0.590435,0.160296,108,1454,1562
6  5,2011-01-05,1,0,1,0,3,1,1,0.226957,0.22927,0.436957,0.1869,82,1518,1600
7  6,2011-01-06,1,0,1,0,4,1,1,0.204348,0.233209,0.518261,0.0895652,88,1518,1606
8  7,2011-01-07,1,0,1,0,5,1,2,0.196522,0.208839,0.498696,0.168726,148,1362,1510
9  8,2011-01-08,1,0,1,0,6,0,2,0.165,0.162254,0.535833,0.266804,68,891,959
10 9,2011-01-09,1,0,1,0,0,0,1,0.138333,0.116175,0.434167,0.36195,54,768,822
```

图 4-1　day.csv 文件内容

该文件遵循常规的.csv 格式，第一行包含描述列名的表头，然后是 731 行观测值，每天一行。那么，这些数据代表什么意义呢？仔细查看 readme.txt 文件，就可以找到每列的定义：

- instant：记录索引（表示从数据集中的第一个观测值起经过的日期）。
- dteday：日期。
- season：季节（1—春季，2—夏季，3—秋季，4—冬季）。
- yr：年（0—2011，1—2012）。
- mnth：月（1～12）。
- holiday：这一天是不是假日（从 http://dchr.dc.gov/page/holiday-schedule 中提取）。
- weekday：周日。
- workingday：非周末且非假日为 1，否则为 0。
- weathersit：。
  - 1：晴，少云，局部多云。
  - 2：雾+多云，雾+碎云，雾+少云，雾。
  - 3：小雪，小雨+雷暴+散云，小雨+散云。
  - 4：暴雨+冰雹+雷暴+雾，雪+雾。
- temp：规格化温度（摄氏度），温度值除以 41（最高温度）。
- atemp：规格化体感温度（摄氏度）。温度值除以 50（最高温度）。
- hum：规格化湿度。湿度值除以 100（最大值）。
- windspeed：规格化风速。风速值除以 67（最大值）。
- casual：临时用户计数。
- registered：注册用户计数。
- cnt：租出的自行车总数，包括临时和注册用户。

我们有几个不同的可用特征：时间/日历信息——常规工作日或者“特殊日期”，多个与天气有关的计量指标，最后是当天租赁自行车的人数，按照注册用户和临时用户分解。

我们的焦点是创建一个模型，利用某一天的可用计量指标（当然，要排除 casual 和 registered，因为它们没有什么意思，从定义上 casual+registered=cnt），预测 cnt——在特定日期内租赁自行车的人数。注意，我们的数据不是同质的：有些特征是数值——也就是真实的数字（例如 temp ——温度），有些则是分类数据——每个数字代表一个状态（例如 weathersit 或 workingday）。

4.1.2 用 FSharp.Charting 检查数据

和往常一样，我们从脚本文件开始，仔细观察数据，用 fsharp.Data 中的 CSV 类型提供程序将其加载到内存中，首先从 NuGet 安装 Fsharp.Data：

```
#I @"packages\"
#r @"FSharp.Data.2.2.1\lib\net40\FSharp.Data.dll"

open FSharp.Data
```

```
type Data = CsvProvider<"day.csv">
let dataset = Data.Load("day.csv")
let data = dataset.Rows
```

和我们前面研究的一些问题不同，这个数据集有一组相对明显的特征可供入手。在这种情况下，人工检查数据以观察是否存在模式是值得的。我们首先绘制一段时间内的自行车使用情况图表。为此，将使用另一个 F#库 fsharp.Charting，该库提供一组经典图表。我们用 NuGet 安装该库并创建第一个图表，绘制总使用量的线图：

```
#load @"FSharp.Charting.0.90.10\FSharp.Charting.fsx"
open FSharp.Charting

let all = Chart.Line [ for obs in data -> obs.Cnt ]
```

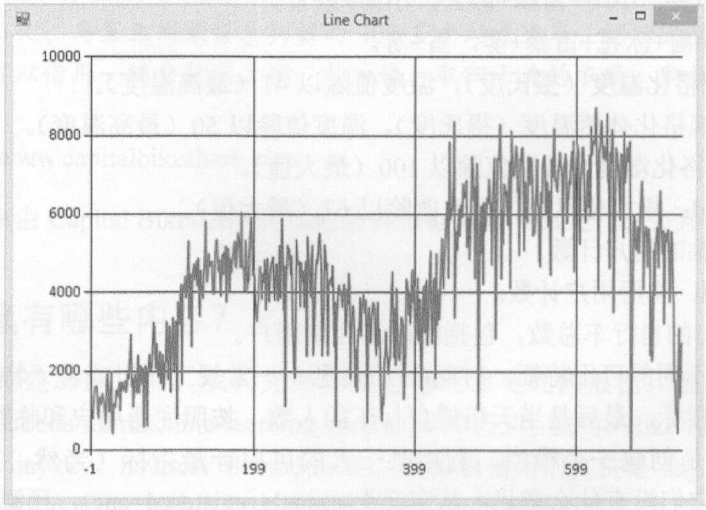


图 4-2　从第一个观测值起的逐日自行车使用量图表

提示： FSharp.Charting 可以作为常规库使用，而且还自带了一些实用工具，使其在交互式脚本环境中显得很友好。特别是，创建的图表（如 Chart.Line [...]）将自动显示，而无须调用.ShowChart()。要使用“交互模式”，只需加载 fsharp.Charting.fsx，以代替 fsharp.Charting 动态链接库。

图表上立刻能看到一些显著的现象。首先，数据相当不规则——每天的波动很大。就这一点而言，波动的幅度很大：使用量在一段时间内经历了高点和低点。最后还能看出，总体的趋势是向上的。

4.1.3　用移动平均数发现趋势

我们能够确认对趋势的直觉吗？方法之一是使用所谓的移动平均数。为了减少逐日数据

中的噪声，可以在一定长度的时间窗口内求取平均值，然后绘制结果曲线，隔离深层次的趋势和短期波动。F#的 Seq 模块包含一个方便的函数 Seq.windowed，该函数取得一个序列并将其转换为包含连续观测值块的新序列：

```
let windowedExample =
    [ 1 .. 10 ]
    |> Seq.windowed 3
    |> Seq.toList
```

上述代码将序列 1～10 聚合为 3 个连续值的块：[[|1; 2; 3|]; [|2; 3; 4|]; [|3; 4; 5|]; [|4; 5; 6|]; [|5; 6; 7|]; [|6; 7; 8|]; [|7; 8; 9|]; [|8; 9; 10|]]。由此，只需计算每块的平均值即可生成移动平均数：

```
let ma n (series:float seq) =
    series
    |> Seq.windowed n
    |> Seq.map (fun xs -> xs |> Seq.average)
    |> Seq.toList
```

我们在原始序列图表上覆盖 7 日和 3 日移动平均数：

```
Chart.Combine [
    Chart.Line count
    Chart.Line (ma 7 count)
    Chart.Line (ma 30 count) ]
```

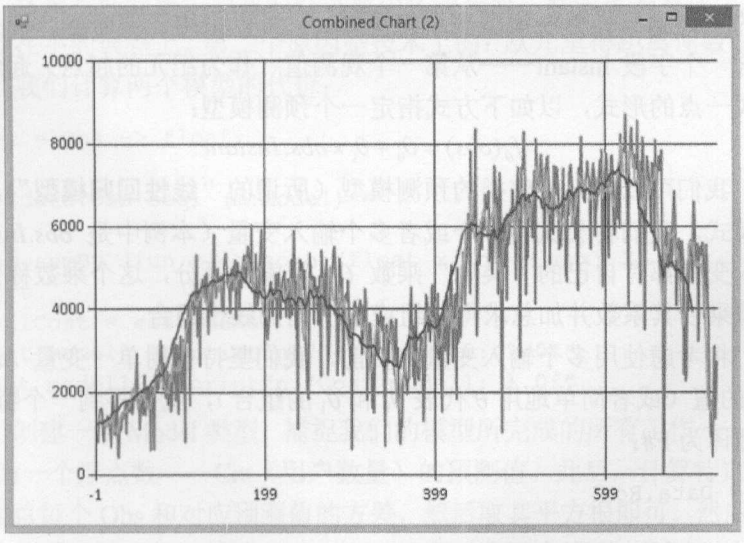


图 4-3　7 日和 3 日移动平均数

移动平均数虽然本身不是决定性的，但是提示了较长期的趋势，清晰地显示了一整年时间内的某些规律，包括季节性的高峰。

4.2　为数据适配模型

我们的前进目标是构建一个模型，根据数据集中已有的信息预测自行车使用情况。在更进一步之前，先进行一些量化工作，确定任务的基准。如何定义结果的好坏？我们所能创建的最简单模型就是始终预测同一个数字——平均数。在那种情况下，典型误差会是多少呢？

```
let baseline =
    let avg = data |> Seq.averageBy (fun x -> float x.Cnt)
    data |> Seq.averageBy (fun x -> abs (float x.Cnt - avg))
```

使用这种原始方法，我们的预测值和正确值的平均偏差为 1581.79——这是必须超越的数字。让我们来看看是否能够做得更好。

4.2.1　定义简单直线模型

可能有效的最简单模型是什么？平均使用量似乎不断增加，因此我们按照这一思路，首先加入趋势，尝试改善原始模型：

$$\text{使用量}(t) = \text{常量} + t \times \text{增长率}$$

换言之，我们将使用量表示为一条直线，从某个起始值开始，随时间推移线性增长。这可能不是我们所能提出的最佳方案——例如，这没有考虑我们已经发现的季节效应——但是有可能比原始的“预测平均值”模型表现得更好。t 可以采用任何时标，但是数据集中的观测值恰好都包含一个字段 instant——从第一个观测值（作为纪元的起点）起经过的天数。我们可以用稍正式一点的形式，以如下方式指定一个预测模型：

$$f_\theta(obs) = \theta_0 + \theta_1 \times obs.Instant$$

也就是说，我们可以定义一整类的预测模型（所谓的**“线性回归模型”**），这些模型都遵循相同的通用模式。我们将尝试用一个或者多个输入变量（本例中是 *obs.Instant*）预测一个数值。每个输入变量都有自己的“类属”乘数 θ，用索引区分，这个乘数称作**回归系数**。预测值由每个变量乘以其系数并加总求得，组成了所谓的**线性组合**。

稍后，我们将考虑使用多个输入变量。目前，我们坚持使用单一变量 *obs.Instant*。如果设置了 θ_0 和 θ_1 的值（或者简单地用 θ 代表 θ_0 和 θ_1 的组合），就会得到一个特定的预测模型。模型可以直接翻译为 F#：

```
type Obs = Data.Row

let model (theta0, theta1) (obs:Obs) =
    theta0 + theta1 * (float obs.Instant)
```

我们以两个随机模型为例，模型 0 中 θ_0 设置为序列的平均值，另一个模型（模型 1）中随机地将 θ_0 设置为 6000，θ_1 设置为–4.5。现在，我们可以根据真实数据绘制两个模型的预测图表：

```
let model0 = model (4504., 0.)
let model1 = model (6000., -4.5)

Chart.Combine [
    Chart.Line count
    Chart.Line [ for obs in data -> model0 obs ]
    Chart.Line [ for obs in data -> model1 obs ] ]
```

显然，两个模型都不能很好地拟合数据。从好处说，我们有了一个通用结构，可以定义广泛的预测模型。但是，有一个新问题：如何为 θ 选择一个“好”的值？

4.2.2 寻找最低代价模型

第一个问题是更加清晰地定义我们的目标。通俗地讲，我们所寻求的是与数据匹配得最好的直线。用稍微不同的方式表达——我们所要的是一条曲线，在与实际观测值相比较时提供可能的最小误差——或者，与实际数据距离最小的曲线。第 1 章中已经见过类似的方法！我们可以简单地改编距离函数，这次计算的不是两个图像的逐点距离，而是预测曲线与实际值的欧几里得距离，比较每个数据点的真值和预测值。

你可能会问，为什么选择欧几里得距离而不是其他距离？毕竟，我们在第 1 章中是从曼哈顿距离开始的，因为它容易计算。这一选择有几个原因。第一个原因是基于建模的考虑。欧几里得距离通过求差值的平方数对误差予以惩罚，因此，较大的误差比较小的误差受到的惩罚严重得多，这是一件好事。当我们尝试最小化距离时，就会主要专注于避免大的误差，而不会将精力放在小的误差上。第二个原因是技术上的：欧几里得距离使数学模型更加简单。

举个例子，我们计算两个模型的代价：

```
type Model = Obs -> float

let cost (data:Obs seq) (m:Model) =
    data
    |> Seq.sumBy (fun x -> pown (float x.Cnt - m x) 2)
    |> sqrt
let overallCost = cost data
overallCost model0 |> printfn "Cost model0: %.0f"
overallCost model1 |> printfn "Cost model1: %.0f"
```

首先，我们创建一个 Model 类型，捕捉我们的模型所完成的所有工作——也就是说，将观测值 Obs 转换为一个浮点数——Cnt（用户数量）的预测值。此后，计算特定数据集的代价很简单，只需要加总每个 Obs 和对应预测值的方差，然后取其平方根即可。然后，我们可以使用部分应用创建计算整个数据集代价的 overallCost 函数，在两个模型上测试，产生如下结果：

```
>
Cost model0: 52341
Cost model1: 71453
```

在本例中，model0 的代价远低于 model1，说明它更适合我们的数据，与图 4-4 一致。现在，我们可以重新陈述问题：我们试图实现的是找出一个 θ 值，使代价函数取值最小。太棒了！我们已经知道想要达到的目标，动手吧！

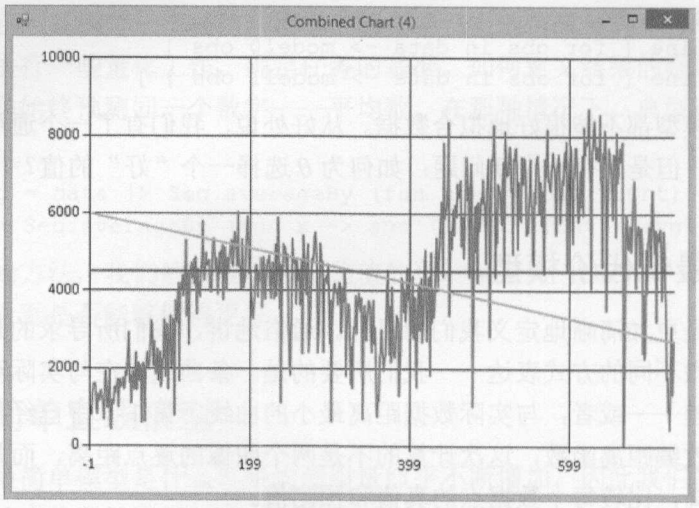


图 4-4　两个线性回归模型示例图解

4.2.3　用梯度下降找出函数的最小值

我们将要使用的方法叫作**梯度下降**，其思路是从一个 θ 值开始，观察预测误差，逐渐调整 θ 以减小误差。为了使用这种方法，我们需要一种手段，取得当前模型并向降低误差的方向移动 θ_0 和 θ_1。

在进入实际问题之前，我们首先复习一下微积分，目标是解释梯度下降的原理。如果你对微积分不太熟悉（或者不感兴趣），没有必要担心——仍然可以应用梯度下降方法。然而，理解其工作原理很有趣。这是一种相当通用的技术，适用于寻找某个函数最大值的场合。它在机器学习中应用特别广泛，可以识别拟合某个数据集的最佳参数，这正是我们接下来要做的。

从一个简单的例子开始：

$$y=f(x),\ f(x)=3\times x-7$$

函数 f 描述了一条直线，其导数为：

$$f'(x)=3$$

这个数字代表什么？导数对应 f 的斜率，换句话讲，x 从当前值增加“少许”时 y 值的增长。这一特例没有太多趣味：在直线上的任何位置，斜率都是恒定的。这也暗示着，f 没有最小值：对于任意值 x，x 减小都将导致 $f(x)$ 减小。然而，考虑一下更有趣的例子：

$$y=g(x),\ g(x)=2\times x^2-8\times x+1$$

函数 g 是一个二次多项式，其导数为：

$$g'(x)=4\times x-8=4\times (x-2)$$

这比上个例子有趣多了，斜率对于每个 x 值都不同。而且，它会改变符号：$x<2$ 时斜率为负数，$x>2$ 时为正数，$x=2$ 时为 0。这意味着，$x>2$ 时，斜率向上：如果我们“稍微”减小 x，斜率将减小，$g(x)$也应该减小。相反，对于低于 2 的值，导数为负，x 的稍许增加将使斜率下降，g 也将减小。

图 4-5 展示了 $x=3$ 时的函数 g 及其导数。$g'(3)=4$：在 $x=3$ 的附近，稍微增加 x，将在 y 上造成 4 倍的增长。

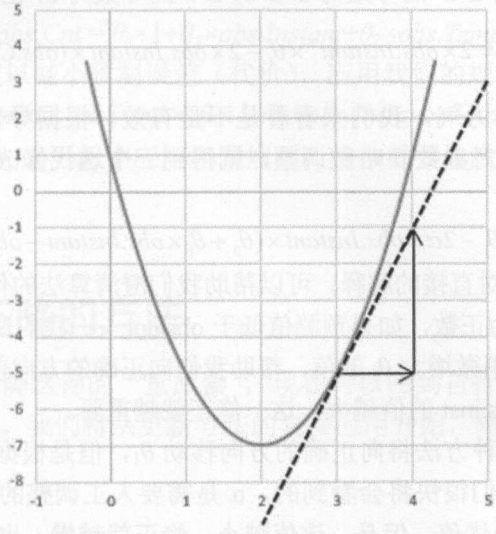


图 4-5 使用导数求出曲线的斜率

$x=2$ 时又是如何？对于该值，斜率为 0，意味着我们到达了一个“平坦”的区域，也就是说：没有明显的去向。导数为 0 的点称作“**极值点**”，可能是最小值或者最大值。

那么，为什么说这很有趣呢？因为我们可以用这种现象设计一个通用策略，寻找函数的最小值。从随机值 x 开始，可以计算导数（或者梯度）逐步迈向下降的方向：

$$x_{k+1}=x_k-\alpha\times g'(x_k)$$

α 的值称作“**学习速率**”。斜率只有在非常接近 x 时才能得到正确值，α 使我们可以调整更新 x 的步长。还要注意，变化取决于 g'。当斜率很大时，我们对 x 进行较大的改变；当 g' 较小时，很有可能接近最小值，我们将做较小的调整。利用这种方法，如果一切按计划进行，就可以得到一个序列 x_0，$x_1\cdots$，每一项都产生更小的 g 值，这就是梯度下降算法。

4.2.4 使用梯度下降进行曲线拟合

下面我们将上述想法应用到手上的问题。我们将稍微调整一下问题：不将焦点放在模型预测的值上，而是考虑我们试图减小的误差（或者代价函数）。另一个调整是，不考虑将观测值 x 作为输入参数，而是换一个角度，考虑以 θ 作为参数。在这个框架下，使用参数 θ 计量模型代价的函数如下：

$$cost\ (\theta_0, \theta_1)=(obs.Cnt-\ (\theta_0+\theta_1\times obs.Instant))^2$$

换言之，这个函数计量对模型使用参数 θ_0 和 θ_1 时，某个观测值 obs 的误差。如果我们稍微增加 θ_1，误差将如何变化？这就是代价对 θ_1 的导数，让我们来计算它。首先，明确各项，将代价改写为 θ_1 的函数，略做扩展并重新组织各项为 a $\times\theta_1^2$ +b $\times\theta_1$+c 的形式：

$$cost\ (\theta_1)=\theta_1^2\times obs.Instant^2-\theta_1\times\ (2\times obs.Instant\times\ (obs.Cnt-\theta_0))+(\ obs.Cnt-\theta_0)^2$$

这是一个简单多项式，a$\times x^2$ 的导数为 2×a×x，b×x 的导数为 b，由此得出代价函数的导数是：

$$\frac{\partial \cos t}{\partial \theta_1}=2\times obs.Instant^2\times\theta_1-2\times obs.Instant\times(obs.Cnt-\theta_0)$$

太棒了！这花了很多力气，我们来看看是不是有效。根据导数的公式“稍微”移动 θ_1 应该减小预测误差。我们对参数 α 略做调整，就得到一个迭代修改 θ_1，在每一步中都减小观测值误差的更新过程：

$$\theta_1\leftarrow\theta_1-2\alpha\times obs.Instant\times(\theta_0+\theta_1\times obs.Instant-obs.Cnt)$$

注意，这一公式有相对直接的解释，可以帮助我们澄清算法的作用。方程中的最后一项就是误差项。假定 obs.inst 为正数，如果预测值低于 obs.cnt——我们试图预测的实际值——模型就是“下冲”的，算法将稍微增大 θ_1 的值，帮助我们向正确的方向前进，如果情况相反，则减小 θ_1 的值。还要注意，obs.inst 的值越大，这一修正就越重要。

最后要注意，虽然这种方法将向正确的方向移动 θ_1，但是根据不同的 α 值，我们可能会得到过度的补偿。正如我们很快将会看到的，α 是需要人工调整的参数。如果 α 足够小，该算法将顺利地得出 θ_1 的最优值。但是，该值越小，修正就越慢，也就意味着这一过程得出 θ_1 最优值的等待时间越长。

4.2.5　更通用的模型公式

对手上的问题应用上述方法之前，趁着还没有忘记相关的数学原理，我们再说明两点，使算法更加通用。想象一下，我们所要尝试的不是之前已经看到的简单直线，而是有 N 个特征 X_1，X_2，…，X_N，试图用如下形式的预测模型进行拟合：

$$Y=\theta_0+\theta_1X_1+\theta_2X_2+\cdots+\theta_NX_N$$

这种模型和我们正在使用的模型类似，只是特征更多。我们的简单直线模型就是 N=1 的特例。第一个方便的技巧是在模型中引入一个“伪”特征 X_0，对每个观测值都设 X_0=1。这样做的原因是，我们在编写预测模型时可以一致的方式对待所有 θ 值，无须对常数项 θ_0 做特殊处理：

$$Y=\theta_0X_0+\theta_1X_1+\theta_2X_2+\cdots+\theta_NX_N$$

结果是，现在可以改写代价函数，对于任何 θ 值，公式如下：

$$cost\ (\theta_k)=(Y-(\theta_0X_0+\theta_1X_1+\theta_2X_2+\cdots+\theta_NX_N))^2$$

这里最好的一点是，如果现在尝试为 0～N 之间的任何 k 值找出更新 θ_k 的方法，在求导并重复上面的步骤之后，最终得到如下公式：

$$\theta_k \leftarrow \theta_k - 2\alpha \times X_k \times (\theta_0 X_0 + \theta_1 X_1 + \theta_2 X_2 + \cdots + \theta_N X_N)$$

很了不起，利用这种方法，现在我们已经有了同时更新所有参数 θ（包括应用到常数项的 θ_0）的算法，可用于任意多个特征。

下面说明如何在我们的模型上利用这一思路——一个实例可能有助于使它看起来不那么抽象。例如，我们可以用天数和温度预测用户数量，这两个变量起到 X_1 和 X_2 的作用。此时，回归模型如下：

$$obs.Cnt = \theta_0 \times 1 + \theta_1 \times obs.Instant + \theta_2 \times obs.Temp$$

如果试图更新 θ_2 的值以减小预测误差（代价），应用我们的规程将得到如下更新规则：

$$\theta_2 \leftarrow \theta_2 - 2\alpha \times obs.Temp \times (\theta_0 \times 1 + \theta_1 \times obs.Instant + \theta_2 \times obs.Temp)$$

而且，可以在 θ_0 和 θ_1 的更新中应用相同规则；唯一需要修改的是替换公式中对应的 θ_k 和 X_k。

4.3 实施梯度下降的方法

我们已经有了梯度下降这样的“重武器”，现在可以回到自行车问题，使用算法通过数据集最终找出适合的直线。我们将从更新方法的直接应用开始，然后分析结果以观察如何改善这一方法。

4.3.1 随机梯度下降

我们可以采用的第一种方法是简单地在观测值上迭代，在每一步中进行小的调整。首先，需要一个函数，在给定的观测值和 θ 值下，以学习速率（α）更新 θ：

```
let update alpha (theta0, theta1) (obs:Obs) =
    let y = float obs.Cnt
    let x = float obs.Instant
    let theta0' = theta0 - 2. * alpha * 1. * (theta0 + theta1 * x - y)
    let theta1' = theta1 - 2. * alpha * x * (theta0 + theta1 * x - y)
    theta0', theta1'
```

这是我们前面所求公式的直接实现，计算 θ 的更新值并以元组形式返回。我们用一个例子确认这种方法有效，例如，取第 100 个观测值，从 θ 的原始值（0.0, 0.0）开始：

```
let obs100 = data |> Seq.nth 100
let testUpdate = update 0.00001 (0.,0.) obs100
cost [obs100] (model (0.,0.))
cost [obs100] (model testUpdate)
```

看起来一切都正常：应用更新之后，特定观测值的新预测得到改善，误差减小。此时，我们需要做的就是将之应用到整个数据集，在每个观测值上从前一步得到的 θ 值入手，逐步更新：

```
let stochastic rate (theta0,theta1) =
    data
    |> Seq.fold (fun (t0,t1) obs ->
        printfn "%.4f,%.4f" t0 t1
        update rate (t0,t1) obs) (theta0,theta1)
```

这里，我们使用 Seq.fold，它等价于 C#LINQ 中的 Enumerable.Aggregate 方法。Fold 函数简单地取得一个初始值和一个值序列，对序列的每个元素应用同一个元素，更新累加器值。一个简单的例子有助于说明这一思路：

```
let data = [0;1;2;3;4]
let sum = data |> Seq.fold (fun total x -> total + x) 0
```

Fold 记录一个总计值，最初为 0，将每个值 x 加到该值中，直到序列中没有剩下任何值——到那个时候，返回总计值。

在我们的例子中，遵循相同的模式：用两个值初始化累加器——θ_0 和 θ_1——对数据集中的每个观测值更新它们，直到没有剩下任何观测值，然后返回 θ 的最终值。

我们已经接近成功。剩下的唯一问题是检验学习速率 α，取一个既足以避免过度调整，又足以避免算法进展太慢的值。这很容易做到，因为我们可以测试不同的 α 值（0.1,0.01,0.001,…），从 θ = (0.0,0.0) 开始，比较数据集上一遍更新之后的结果质量：

```
let tune_rate =
    [ for r in 1 .. 20 ->
        (pown 0.1 r), stochastic (pown 0.1 r) (0.,0.) |> model |> overallCost ]
```

打印输出每个学习速率的代价，得到如下结果：

```
Alpha                 Cost
0.1                 : NaN
0.01                : NaN
0.001               : NaN
0.000,1             : NaN
0.000,01            : Infinity
0.000,001           : 94,007
0.000,000,1         : 93,189
0.000,000,01        : 59,367
0.000,000,001       : 107,780
0.000,000,000,1     : 129,677
0.000,000,000,01    : 132,263
0.000,000,000,001   : 132,526
0.000,000,000,000,1 : 132,552
0.000,000,000,000,01 : 132,555
0.000,000,000,000,001: 132,555
```

结果很典型，对于较大的 α 值，调整过于激进，以至于 θ 变动很大，无法得到稳定的结果。相反，较小的值导致调整幅度很小，在改进拟合程度上几乎毫无进展。最有效的 α 值为 1e-8（0.00000001），对于 θ 值（0.03, 8.21），其代价为 59367。绘制结果曲线，可以得到如下结果（见图 4-6）：

```
let rate = pown 0.1 8
let model2 = model (stochastic rate (0.0,0.0))

Chart.Combine [
    Chart.Line count
    Chart.Line [ for obs in data -> model2 obs ] ]
```

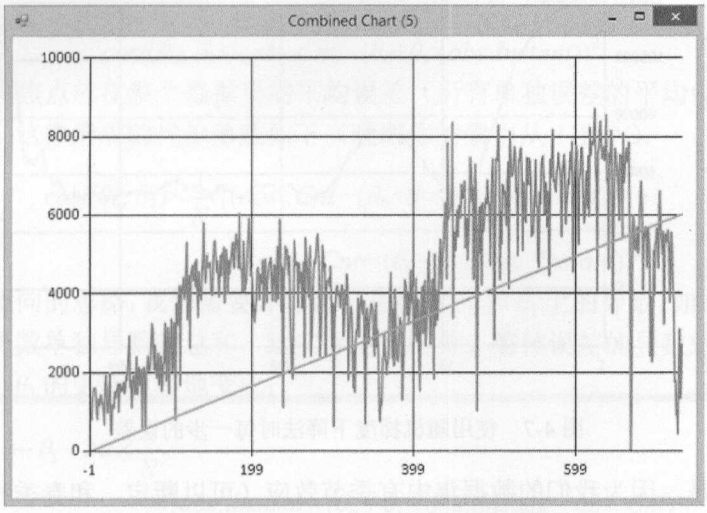


图 4-6 使用随机梯度下降得到的最佳匹配曲线

■ **注意**：我们发现了一个具有合理效果的 α 值，但是还有另一个微妙的问题。曲线的斜率看起来不错，但是起始值 θ_0 似乎太小。问题是我们的变量采用了不同的标度：常量为 1，但是 instant 的范围是 0 到大约 700 之间。因此，调整主要是由具有较大量级的特征 θ_1 推动的。为了更快地调整 θ_0，我们必须准备数据集，使所有输入处于可比较的量级——这一过程称作重新标度。现在，我们忽略该问题，但是将在第 5 章中详细讨论特征重新标度。

4.3.2 分析模型改进

现在该怎么办？下一步是不断重复过程，在数据集上进行多遍迭代，直到观察不到明显的改进。但是，我们不这样做，而是进一步观察算法的表现，具体地说，是观察预测误差的变化。代码的细节并不重要——在这里，我们要做的是取一个激进的 α 值（10 倍于前面确定的“最有效点”），绘制执行每次调整之后的总体误差图表：

```
let hiRate = 10.0 * rate
let error_eval =
    data
    |> Seq.scan (fun (t0,t1) obs -> update hiRate (t0,t1) obs) (0.0,0.0)
    |> Seq.map (model >> overallCost)
    |> Chart.Line
```

这很有趣，在慢速启动之后，误差开始稳步下降，在中段下降的速度减慢，最后又开始爬升。发生了什么？采用这个激进的学习速率，得到了图 4-7 所示的图表。

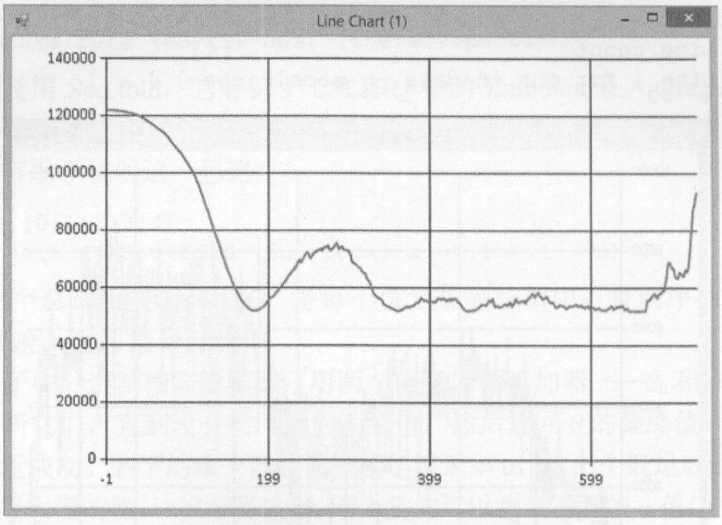


图 4-7　使用随机梯度下降法时每一步的误差

发生的情况是，因为我们的数据集中有季节效应（可以断定，和春季相比，冬季骑自行车上班的想法吸引力较差），多个连续的观测值展现出相同的趋势（在理想线之上或者之下）。因此，更新过程一次仅关注一个观测值，进行的一系列修正都向相同方向倾斜（因为误差都向相同方向倾斜），从而与实际情况渐行渐远，暂时增加了整体模型的误差。

这有几个方面的意义。首先，除了梯度下降，如果我们使用交叉验证评估工作质量，必须非常小心。例如，如果以最后 200 个观测值作为验证样本，可能得到对预测误差非常偏颇的估计，因为在这个特定的时间段内，曲线呈现下降趋势，而这只是暂时现象，当然不能代表数据的整体情况。

这说明了随机下降方法的潜在局限性。用通俗的话讲，我们尝试实现一致的估算，但是处理过程实际上是一个接一个地探查每个观测值，并遵循最后一个观测值表现出来的方向。这可能导致某些不必要的波动，例如，遇到一个或者一连串的非典型观测值。公平地说，这个问题可以通过随机而非顺序选取观测值（这将限制连续观测值表现出类似偏差的风险）和激进地降低 α 值（这将降低任何单独观测值的权重，降低更新过程速度）缓解——但是即使如此，这一方法仍然存在波动的可能性。

处理上述问题的方法之一是避免依赖单个观测值，而代之以根据整个数据集进行更新。这是批量梯度下降的基础，我们将在下面讨论这种方法的实际运用。与此同时，不应该完全抛弃随机梯度下降，该方法易于实施，适合于广泛的情况。更有趣的是，它可以进行所谓的“在线学习”。该算法在学习时不需要事先获得整个数据集，如果随着时间的推移获得新数据，你的模型可以仅用当前模型和最新的观测值进行更新。

4.3.3 批量梯度下降

那么，如何改造我们的模型，根据整个数据集（而不是依赖单独观测值）进行更新？事实上，这相当容易完成。在随机下降方法中，我们的目标是最小化单一观测值上的误差，这由代价函数定义：

$$cost(\theta_0, \theta_1)=(obs.Cnt-(\theta_0+\theta_1\times obs.Instant))^2$$

我们也可以将焦点放在整个数据集的平均误差（所有单独误差的平均值）上，而不是聚焦于单一观测值。这样形成的代价函数如下（观测值的索引从 1 到 N）：

$$cost(\theta_0, \theta_1)=\frac{1}{N}[(obs_1.Cnt-(\theta_0+\theta_1\times obs_1.Instant))^2+\cdots$$
$$+(obs_N.Cnt-(\theta_0+\theta_1\times obs_N.Instant))^2]$$

根据和前面相同的思路，我们需要计算代价在参数 θ_0 和 θ_1 上的导数。如果你记得 $f(x)+g(x)$ 的导数等于两个函数单独导数的总和，这实际上很简单；整体误差的导数就是每个单独观测值上导数的总和。θ_1 的更新简单地变成：

$$\theta_1 \leftarrow \theta_1-2\alpha\times\frac{1}{N}$$
$$\times[obs_1.Instant\times(\theta_0+\theta_1\times obs_1.Instant-obs_1.Cnt)+\cdots$$
$$+obs_N.Instant\times(\theta_0+\theta_1\times obs_1.Instant-obs_N.Cnt)]$$

解释上述函数的方法之一是，该算法不根据单独观测值的误差更新，而是计算每个观测值上的调整，然后取平均修正值。这很容易以与随机下降时相同的方式扩展到所有 θ。因为我们希望拯救森林，所以这次不重复所有的步骤，而是采用经典的做法“我们将把这个问题作为读者们的练习，让大家自己深入细节”，直接跳到更新算法的一种可能实现：

```
let batchUpdate rate (theta0, theta1) (data:Obs seq) =
    let updates =
        data
        |> Seq.map (update rate (theta0, theta1))
let theta0' = updates |> Seq.averageBy fst
let theta1' = updates |> Seq.averageBy snd
theta0', theta1'
```

我们要做的是取得每个观测值，根据该观测值单独计算 θ 的调整值，再取所有调整的平均值。然后重复更新过程，逐步微调 θ 的估算值，在一定的迭代次数之后或者搜索已经稳定、θ 值在两次迭代之间没有太大变化时停止：

```
let batch rate iters =
    let rec search (t0,t1) i =
        if i = 0 then (t0,t1)
        else
```

```
            search (batchUpdate rate (t0,t1) data) (i-1)
search (0.0,0.0) iters
```

我们只是遵循和前面一样的步骤，找出学习速率 α 的合适值。对于 α=0.000001，如果将误差作为迭代的函数作图，可以得到一个漂亮而平滑的曲线，说明误差不断降低，没有任何波动（见图 4-8）。看看如下的代码和结果：

```
let batched_error rate =
    Seq.unfold (fun (t0,t1) ->
        let (t0',t1') = batchUpdate rate (t0,t1) data
        let err = model (t0,t1) |> overallCost
        Some(err, (t0',t1'))) (0.0,0.0)
    |> Seq.take 100
    |> Seq.toList
    |> Chart.Line

batched_error 0.000001
```

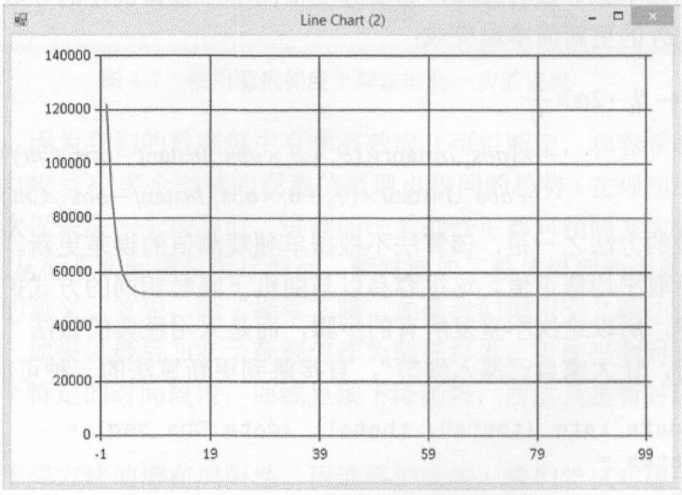


图 4-8　使用批量梯度下降法时每步的误差

这样的结果比前面好多了。一方面，我们现在一次使用整个数据集，在目前的情况下不成问题，但是如果数据集很大就会有问题。另一方面，我们的规程现在稳定地收敛于最优值，没有任何反复：等待的时间越长，估算的结果越好。最好的是，同一个方法也适用于以直线拟合数据之外的情况。实质上，只要你的代价函数可导且是凸函数，就可以采用同样的通用方法。

然而，我们的方法仍然有小缺点。首先，人工调整学习速率令人烦恼——如果不需要这么做就好了。其次，如果大量迭代运行该算法，就会注意到代价不断降低，参数 θ_0 不断增加，只有经过很长的时间才能最终稳定。这是因为前面提到的问题：θ_0 和 θ_1 使用的标度差别很大，如果学习速率较慢，算法必须经过大量迭代才能求出“正确值”。在下一小节中，我们将会看

到，对于手上的具体问题（没有任何规范化惩罚因子的线性回归），可以使用少数线性代数知识简化算法，使之大大加速。

4.4 拯救者——线性代数

到目前为止，我们的方向紧跟问题域，将数据集表示为一组具有命名属性的观测值，用特定于问题域的映射和分析应用转换。然而，想想就知道，一旦开始应用更新过程，域的细节就不再重要了。例如，如果我们试图将车的价格作为发动机和里程的函数进行预测，而不是将自行车使用量作为风速的函数预测，方法没有什么不同。最终，在两种情况下，我们都将观测值归纳为双精度数组成的数据行，用它们预测另外一个双精度数。

在这个框架下，问题域的细节并不重要，将重点放在数据的结构上，把问题域转换成公共结构，应用通用算法解决共享同一结构的问题，是更有意义的。这就是线性代数带来方便的地方：它为我们提供了一种明确的语言，可以描述多行数字以及在它们之上进行不同运算的方法。

线性代数复习

线性代数的两个基本要素是向量和矩阵。n 阶向量就是 n 个元素（索引从 1 到 n）的一个集合，如：

$$x = [1.0 \quad 5.0 \quad -2.0] = [x_1 \quad x_2 \quad x_3]$$

矩阵以其维度（或者大小）——$m \times n$ 描述，其中 m 指的是行数，n 是列数。举个例子，下面的 $\boldsymbol{M}$ 是一个 3×2 矩阵：

$$\boldsymbol{M} = \begin{bmatrix} x_{1,1} & x_{2,2} \\ x_{2,1} & x_{2,2} \\ x_{3,1} & x_{3,2} \end{bmatrix}$$

作为宽泛的类比，我们可以将向量和矩阵描述为数组和二维数组的数学等价物。

向量可以表示成行向量或者列向量；这种差别主要在处理矩阵时很重要。矩阵可以视为一组行向量或者列向量，向量上的大部分运算可以视为矩阵运算的特例。

我们需要的 4 种核心运算是加、标量乘、转置和乘。矩阵（或者向量）加就是将相同索引的元素相加，要求两个矩阵的大小完全相同：

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} + \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} + \begin{bmatrix} 9 & 8 \\ 10 & 12 \end{bmatrix}$$

标量乘描述了一个矩阵（或者向量）乘以一个数字（标量），结果是每个元素乘以该标量：

$$2 \times \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 2 & 4 \\ 6 & 8 \end{bmatrix}$$

矩阵转置以 T 表示，改变矩阵的方向，使行变成列，反之亦然：

$$\begin{bmatrix}1 & 2\\ 3 & 4\\ 5 & 6\end{bmatrix}^{\mathrm{T}}=\begin{bmatrix}1 & 3 & 5\\ 2 & 4 & 6\end{bmatrix}$$

最后，矩阵乘的计算是第一个矩阵的每个行向量与第二个矩阵的每个列向量的点乘（或称内积）。两个向量的点乘得到一个数字，是每一对应项乘积的总和：

$$[1\ 3\ -5]\cdot[4\quad -2\quad -1]=1\times 4+3\times(-2)+(-5)\times(-1)=3$$

应用到矩阵，得到如下结果：

$$\begin{bmatrix}行_1\\ \cdots\\ 行_m\end{bmatrix}\times[列_1\cdots列_n]=\begin{bmatrix}行_1\cdot 列_1 & & 行_1\cdot 列_n\\ \cdots & & \cdots\\ 行_m\cdot 列_1 & & 行_m\cdot 列_n\end{bmatrix}$$

注意，为了定义的正确，这一方法要求第一个矩阵的列数与第二矩阵的行数相等。

4.4.1　宝贝，我缩短了公式！

让我们来看看如何以代数的方式重新叙述前面的问题。之前，我们的预测模型定义为：

$$\boldsymbol{Y}=\theta_0 X_0+\theta_1 X_1+\theta_2 X_2+\cdots+\theta_N X_N$$

如果将 $\boldsymbol{Y}$ 定义为 obs.cnt，$\boldsymbol{\theta}$ 和 $\boldsymbol{X}$ 定义为两个向量：

$$\boldsymbol{\theta}=[\theta_0;\ \theta_1;\ \theta_2;\cdots;\theta_N]$$

$$\boldsymbol{X}=[X_0;X_1;X_2;\cdots;X_N]$$

那么，预测模型可以更加紧凑的方式重述，即两个向量的乘积：

$$\boldsymbol{Y}=\theta\times\boldsymbol{X}$$

类似地，在批量梯度下降算法中，我们试图最小化的代价函数是 $\boldsymbol{N}$ 个观测值方差的平均值：

$$cost\ (\theta_0,\theta_1)=\frac{1}{N}[(obs_1.Cnt-(\theta_0+\theta_1\times obs_1.Instant))^2+\cdots$$

$$+(obs_N.Cnt-(\theta_0+\theta_1\times obs_N.Instant))^2]$$

同样，可以用 $\boldsymbol{Y}_n$ 代表第 n 个观测值的输出值（即 $\boldsymbol{Y}$ 向量的第 n 个元素），$\boldsymbol{X}_n$ 代表描述第 n 个观测值的数据行，以更加紧凑的方式重新描述上述公式：

$$cost(\theta)=\frac{1}{N}[(Y_1-\theta\times X_1)^2+\cdots+(Y_N-\theta\times X_N)^2]$$

更好的做法是，如果我们将所有观测值堆积起来，可以得到一个矩阵 $\boldsymbol{X}$，其中 $\boldsymbol{X}_n$ 是矩阵的第 n 行，包含第 n 个观测值：

$$cost(\theta)=\frac{1}{N}[\boldsymbol{Y}-\theta\times\boldsymbol{X}^{\mathrm{T}}]\times[\boldsymbol{Y}-\theta\times\boldsymbol{X}^{\mathrm{T}}]$$

我们可以这样继续下去，将整个批量梯度下降算法改写为一系列向量和矩阵预算，在这样做之前，我们先确认算法确实可行，在使用线性代数时，结果确实和原始公式完全一样。

4.4.2 用 Math.NET 进行线性代数运算

虽然有其他选择，但是.NET 上使用线性代数方法的最佳起点是 Math.NET。Math.NET 是一个维护得很好的开源程序库，在出现的数年中经受了考验。我们不再像以前那样在同一个脚本中挤进很多代码，而是启动一个新脚本，通过 NuGet 导入 Math.NET 及其 F#扩展，以及到目前为止使用的其他软件包。首先，看看如何使用 Math.NET 进行基本的线性代数运算：

```
#I @"packages\"
#r @"FSharp.Data.2.2.1\lib\net40\FSharp.Data.dll"
#load @"FSharp.Charting.0.90.10\FSharp.Charting.fsx"
#r @"MathNet.Numerics.Signed.3.6.0\lib\net40\MathNet.Numerics.dll"
#r @"MathNet.Numerics.FSharp.Signed.3.6.0\lib\net40\MathNet.Numerics.FSharp.dll"

open FSharp.Charting
open FSharp.Data
open MathNet
open MathNet.Numerics.LinearAlgebra
open MathNet.Numerics.LinearAlgebra.Double
```

为了演示的目的，我们创建向量 $\boldsymbol{A}$ 和矩阵 $\boldsymbol{B}$，并在它们之上进行一些运算：

```
let A = vector [ 1.; 2.; 3. ]
let B = matrix [ [ 1.; 2. ]
                 [ 3.; 4. ]
                 [ 5.; 6. ] ]
let C = A * A
let D = A * B
let E = A * B.Column(1)
```

一切似乎都符合预期，额外的好处是，代码和打开数学书时看到的数学公式很类似。现在，我们以“代数风格”改写预测和代价函数：

```
type Data = CsvProvider<"day.csv">
let dataset = Data.Load("day.csv")
let data = dataset.Rows

type Vec = Vector<float>
type Mat = Matrix<float>

let cost (theta:Vec) (Y:Vec) (X:Mat) =
    let ps = Y - (theta * X.Transpose())
    ps * ps |> sqrt
```

```
let predict (theta:Vec) (v:Vec) = theta * v

let X = matrix [ for obs in data -> [ 1.; float obs.Instant ]]
let Y = vector [ for obs in data -> float obs.Cnt ]
```

提示：为了简单，我们创建两个类型 Vec 和 Mat，对应于 Vector<float> 和 Matrix<float>。这能很方便地简化我们的代码：Math.NET 支持在数值（Numeric）类型上（而不是浮点和双精度类型）进行代数运算，这可能很实用，但是也可能使代码混乱，需要附加的类型注释澄清预算中的特定类型。考虑到我们打算只在浮点数上运算，保持通用性没有任何附加的好处，只会带来噪声，因此进行这种简化。

现在，可以用相同的 θ 值，将前面获得的预测值和代价与新公式的结果比较：

```
let theta = vector [6000.; -4.5]

predict theta (X.Row(0))
cost theta Y X
```

所有结果都相符。我们有了安全的基础，可以继续估算 θ 了。

4.4.3　标准形式

我们可以“代数风格”改写批量梯度下降程序。然而，我们将采用不同的路径，强调使用代数方法的一些好处。碰巧，我们所要解决的问题（找出代价最小的 θ 值）有数学家所称的“封闭解”：也就是，可从输入中直接计算（无须任何数字逼近方法）的明确、精确解。

解释结果的产生方式超出了本书的范畴，对理解机器学习没有更多的价值，因此我将简单地陈述结果。如果你对此感到好奇，可以在喜欢的搜索引擎中搜索“标准型回归”（normal form regression”。不管理由为何，如下问题

$$\min cost(\theta)=\frac{1}{N}[\boldsymbol{Y}-\theta\times\boldsymbol{X}^{\mathrm{T}}]\times[\boldsymbol{Y}-\theta\times\boldsymbol{X}^{\mathrm{T}}]$$

有一个解 θ：

$$\theta=[\boldsymbol{X}^{\mathrm{T}}\times\boldsymbol{X}]^{-1}\times\boldsymbol{X}^{\mathrm{T}}\boldsymbol{Y}$$

这就是线性代数库大显身手的场合。库中已经为我们实现了矩阵转置和求逆运算，我们的算法现在只需要一行代码。更好的是，我们没有必要经历调整学习速率或者运行数千次代价昂贵的迭代、希望接近正确值的冗长过程——可以一次性获得正确解。观察如下代码：

```
let estimate (Y:Vec) (X:Mat) =
    (X.Transpose() * X).Inverse() * X.Transpose() * Y
```

这是否说明，前面在梯度下降方法上所做的工作都是浪费时间？当然不是！除了通过练习澄清一般过程的教育价值之外，梯度下降是适用于各种情况的通用算法。例如，我们可以对导数做相应的修改，将这种方法用于不同的非线性代价函数。相比之下，标准形式是非常特殊的方程的一种解法，不能随意推广。不过，它所能解决的问题也相当广泛。

4.4.4 利用 MKL 开足马力

到目前为止，对于以代数为中心的方法，我们强调的主要好处是，对于合适的问题，它可以得到更加紧凑的公式。代数成为机器学习的一个重要主题还有一个完全不同的原因——性能。

如果仅用于机器学习，代数在软件中可能不很重要。所幸的是，在另外一个领域中，开发人员非常关心代数，这就是游戏行业。3D 图形到处都涉及多边形的移动，这从根本上就是一个代数问题。因此，归功于对更快、更细致游戏世界的不懈追求，人们在高性能代数运算上投入了巨大的努力，包括硬件层面上的改进。该领域最明显的例子就是 GPGPU（在图形处理单元上的通用计算）——通过原来为矢量图形渲染而设计的硬件实现某类计算速度的极大提升。

GPGPU 的威力巨大，但是也需要进行一些额外的工作。然而，Math.NET 为你提供了利用类似能力的选择，而且不需要任何额外的工作。该库支持一个 MKL 提供程序，该机制实际上允许将计算发送给 **MKL（Math Kernel Library，数学核心库）**——内建于 Intel 处理器的一组高度优化的数学程序库。

在 Math.NET 中，使用 MKL 相当简单。在项目中添加对应的 NuGet 包之后，将出现两个库：MathNet.Numerics.MKL.dll 和 libiomp5md.dll，用鼠标右键单击 MKL dll，然后单击“属性”（Properties），将“复制到输出目录”（Copy to Output Directory）项修改为“总是复制”（Copy Always）。

> **提示：** MKL 有多种风格：32 位和 64 位，Windows 和 Linux。安装 NuGet 包时一定要选择适合你的机器的软件包版本。

此时，将计算发送到 MKL 很简单：

```
System.Environment.CurrentDirectory <- __SOURCE_DIRECTORY__

open MathNet.Numerics
open MathNet.Numerics.Providers.LinearAlgebra.Mkl
Control.LinearAlgebraProvider <- MklLinearAlgebraProvider()
```

第一行有些“丑陋”，其目的是将当前目录设置成源代码所在目录。为了使脚本能够选取位于该目录的 MKL dll，这一行是必需的。完成这一操作之后，剩下的就只是用 MKL 等价物替代默认的 LinearAlgebraProvider，一切就准备就绪了。

那么，这能给我们带来什么呢？根据你所使用的计算机和处理问题的规模不同，效果也不一样。一般来说，对于较小的矩阵（如我们现在要处理的矩阵），MKL 的好处可以忽略。然而，如果在较大的数据集上运算，那么发现 10 倍的加速也是不奇怪的。

4.5　快速演化和验证模型

我们从梯度下降这样的“轻武器”转向标准形式的强大工具。为使模型拟合数据集突然变得相当简单而普通，只要将观测值转换成一个向量就可以了。在本节中，我们将重新编写代码，从通用算法中获益，目标是使我们使用数据做出预测的方法尽可能容易修改，更容易添加或者删除特征，试验不同模型。但是，如果想要创建多个模型，首先需要一个比较质量的过程。我们首先准备这一过程，一旦有了基础，就可以探索如何快速演化和改进模型了。

4.5.1　交叉验证和过度拟合

使用交叉验证作为模型比较的基础应该已经不足为奇了，我们将留下数据集的一部分用于验证，仅用 70%数据训练模型，30%留作验证目的。

注意，这在我们的例子中特别重要。考虑如下情况：假定你考虑使用一组特征$[\theta_0; \theta_1;\cdots;\theta_N]$和一个备选模型$[\theta_0; \theta_1;\cdots; \theta_N; \theta_M]$——也就是说，同一个模型加上一个额外的特征，第二个模型造成的误差是否会比第一个更大？

答案是，第二个特征集包含第一个模型的所有特征和附加特征，其误差总是低于第一个模型。作为非正式的证明，考虑一下：在可能的最糟糕情况下，θ_M为 0，这种情况下两个模型等价，显然误差相同。因此，在两种情况下，搜索最小可能代价的算法所达到的水平相同。然而，在第二种情况下，一旦达到最优值，存在将θ_M从 0 改成另一个数值，改变其他θ达到更低代价的选择。

换一种方式，我们添加的特征越多，代价就越低，不管这些额外的特征是好是坏。例如，如果机械地添加一个完全随机的特征——每天东京一磅红萝卜的价格，我将会得到更好的拟合，或者至少是同样好的拟合。

结论很简单：在训练集上得到好的拟合并不总是好事。同样，我们所要追求的并不是完美拟合所提供数据的模型，而是能够对前所未见的新数据产生可靠预测的模型。

另外，我们还要处理一个额外的问题，如果简单地分割数据集，使用前 70%的观测值进行训练，后 30%进行验证，我们的验证集可能对于未来得到的随机观测值没有“代表性”。因为明显的季节模式，最后一个数据块聚集了类似的观测值，完全忽略了某些情况。为了避免这个问题，我们将首先使用如下的经典 Fisher-Yates 随机洗牌算法，打乱样本以摆脱季节效应：

```
let seed = 314159
let rng = System.Random(seed)
```

```
// Fisher-Yates shuffle
let shuffle (arr:'a []) =
    let arr = Array.copy arr
    let l = arr.Length
    for i in (l-1) .. -1 .. 1 do
        let temp = arr.[i]
        let j = rng.Next(0,i+1)
        arr.[i] <- arr.[j]
        arr.[j] <- temp
arr
```

洗牌算法简单地取一个泛型数组，返回一个包含相同元素，但是以随机顺序重排的新数组：

```
let myArray = [| 1 .. 5 |]
myArray |> shuffle
>

val myArray : int [] = [|1; 2; 3; 4; 5|]
val it : int [] = [|4; 1; 5; 2; 3|]
```

提示：使用随机数生成器（如 System.Random）时，指定种子往往是个好主意。这样，可以在不同会话期间重复结果，有助于确保算法的表现达到你的预期——如果每次运行代码结果都变化，这一点就难以达到！

现在，我们可以简单地打乱数据集并返回两个“切片”，创建训练和验证集，第一个切片占 70%，作为训练集，剩下的 30%作为验证集：

```
let training,validation =
    let shuffled =
        data
        |> Seq.toArray
        |> shuffle
    let size =
        0.7 * float (Array.length shuffled) |> int
    shuffled.[..size],
    shuffled.[size+1..]
```

4.6.2 简化模型的创建

我们已经准备就绪，具备了评估的手段。此时，我们想要的是通过包含或者删除特征，简单地创建多个模型，然后求出有效的参数值。我们需要的是一种将观测值转换成一组特征的手段，每个特征从一个观测值中提取一个浮点数，这样就能将观测值转换为一个特征矩阵，通过标准形式估算对应的 θ。为了实现这一方法，我们创建新类型 Featurizer（特征化器），取得一个 Obs，将其转换为一个浮点数列表：

```
type Obs = Data.Row
type Model = Obs -> float
type Featurizer = Obs -> float list
```

利用这个方法，我们就可以列出用“特征化器”从观测值中提取数据的方式，简化定义模型的手段。举个例子，下面我们将重建简单直线模型，该模型中有一个总是取值为 1 的恒定“伪”变量，从观测值中提取 Instant 属性：

```
let exampleFeaturizer (obs:Obs) =
  [ 1.0;
    float obs.Instant; ]
```

为了方便，我们还创建一个 predictor 函数，该函数以 Featurizer 和向量 theta 为参数，返回将单独观测值转换为预测值的一个函数：

```
let predictor (f:Featurizer) (theta:Vec) =
    f >> vector >> (*) theta
```

与此同时，我们还要创建一个函数，评估给定数据集下特定模型的质量。那样，我们就能在训练集和验证集上比较不同模型。我们当然可以使用代价函数完成评估，但是这里将使用另一个经典指标——平均绝对误差（MAE），因为它更容易掌握。MAE 表明预测的平均偏离程度：

```
let evaluate (model:Model) (data:Obs seq) =
    data
    |> Seq.averageBy (fun obs ->
        abs (model obs - float obs.Cnt))
```

现在将上述功能组合起来：

```
let model (f:Featurizer) (data:Obs seq) =
    let Yt, Xt =
        data
        |> Seq.toList
        |> List.map (fun obs -> float obs.Cnt, f obs)
        |> List.unzip
    let theta = estimate (vector Yt) (matrix Xt)
    let predict = predictor f theta
    theta,predict
```

Model 函数以 Featurizer 类型变量 f（表示希望从观测值中提取的特征）和用于训练模型的数据集作为参数。首先，我们从数据集提取输出 $\boldsymbol{Yt}$ 和矩阵 $\boldsymbol{Xt}$。然后，用 $\boldsymbol{Yt}$ 和 $\boldsymbol{Xt}$ 估算 θ 并返回向量 theta，以及对应的 predictor 函数，该函数此时已经准备就绪。

4.5.3 在模型中添加连续特征

我们在简单直线模型（称作 model0）上测试算法：

```
let featurizer0 (obs:Obs) =
    [    1.;
         float obs.Instant; ]

let (theta0,model0) = model featurizer0 training
```

又简单又好！现在，我们可以在训练和验证集上评估 model0 的质量：

```
evaluate model0 training |> printfn "Training: %.0f"
evaluate model0 validation |> printfn "Validation: %.0f"
>
Training: 1258
Validation: 1167
```

将模型应用于真实数据的结果可视化（见图 4-9）上，可以看到一条漂亮的直线，看上去是相当合理的拟合：

```
Chart.Combine [
    Chart.Line [ for obs in data -> float obs.Cnt ]
    Chart.Line [ for obs in data -> model0 obs ] ]
```

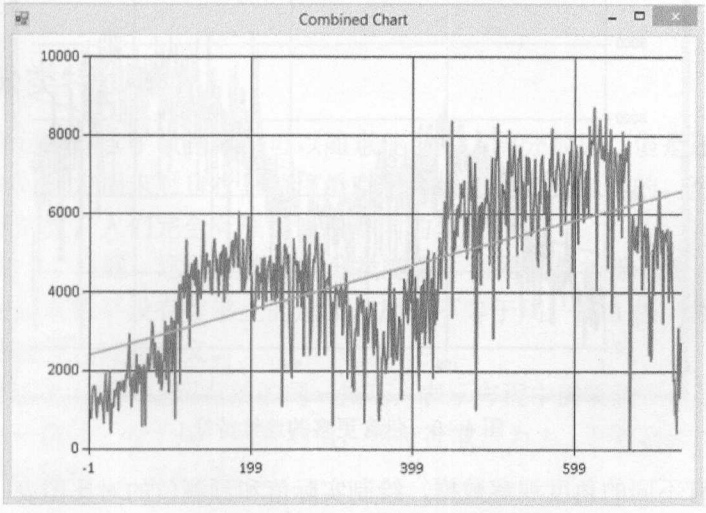


图 4-9 直线回归的可视化结果

现在，我们可以开始在模型中添加选自可用连续变量的特征——也就是有数字意义的变量，如温度或风速——并将其添加到新的特征化器中：

```
let featurizer1 (obs:Obs) =
    [   1.
        obs.Instant |> float
        obs.Atemp |> float
        obs.Hum |> float
        obs.Temp |> float
        obs.Windspeed |> float
    ]

let (theta1,model1) = model featurizer1 training

evaluate model1 training |> printfn "Training: %.0f"
evaluate model1 validation |> printfn "Validation: %.0f"
>
Training: 732
Validation: 697
```

很明显，我们做的是正确的事：在训练集（正如预想）和验证集上（这才是重要的），MAE 都明显下降。之前，我们平均偏离 1200 个单位，现在下降到 750 左右。结果并不完美，但是改善很显著！如果绘制一个新的预测曲线，这个结果很容易看出——新的预测曲线与目标贴得更近（见图 4-10）。

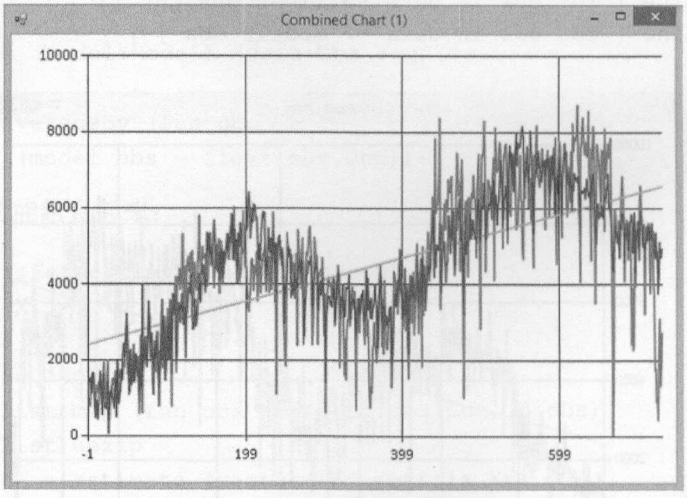


图 4-10　包含更多的连续特征

让我们从稍有不同的角度观察数据，绘制实际值和预测值的对比图表：

```
Chart.Point [ for obs in data -> float obs.Cnt, model1 obs ]
```

图 4-11 确认，我们的模型看上去和真实值更为相近。散点图仍不完美，大约沿着 45 度的直线排列，顶部散布一些噪声点。

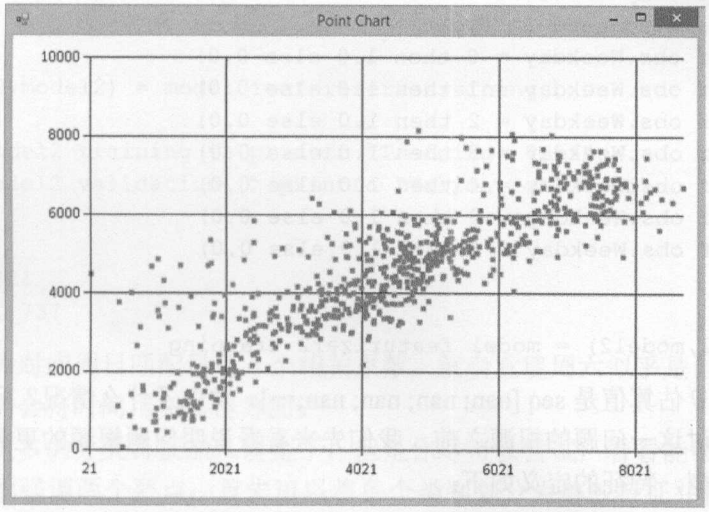


图 4-11 预测值和真实值的散点图

4.6 用更多特征改进预测

现在，我们已经实施了一个结构，可以轻松地从模型中添加或者删除数据，自由地试验，且最大限度地减少了代码的更改。这种方法非常有效，但是数据集中还有许多未曾使用的数据。在本节中，我们将探索更进一步地在模型中包含更多特征、以改进结果的方法。

4.6.1 处理分类特征

我们的模型中已经包含了数据集中可以随意使用的大部分数据，但是还遗漏了一些。例如，假日有何影响？假定周末对自行车使用量有所影响，是完全合理的。至于每周中的各个日子，比如周一和周五，人们完全有理由出现不同的表现。

但是，这里有一个问题：这些变量都是**分类变量**。虽然周日被编码为 0,1,…,6，以代表周日到周六，但是这些数字没有数学上的意义。周三不等于周一的 3 倍，将自行车使用量作为周日编码倍数的模型也没有意义。

但是，我们可以使用已经证明有效的同一技巧，表示方程中的常数项。如果我们为每个周日创建一个特征——例如，对于周一的观测值，该特征取值为 1，否则取值为 0——对 θ 的估算就会捕捉到周一的影响。让我们来试试：

```
let featurizer2 (obs:Obs) =
    [   1.
        obs.Instant |> float
        obs.Hum |> float
```

```
        obs.Temp |> float
        obs.Windspeed |> float
        (if obs.Weekday = 0 then 1.0 else 0.0)
        (if obs.Weekday = 1 then 1.0 else 0.0)
        (if obs.Weekday = 2 then 1.0 else 0.0)
        (if obs.Weekday = 3 then 1.0 else 0.0)
        (if obs.Weekday = 4 then 1.0 else 0.0)
        (if obs.Weekday = 5 then 1.0 else 0.0)
        (if obs.Weekday = 6 then 1.0 else 0.0)
    ]

let (theta2,model2) = model featurizer2 training
```

啊？返回的 θ 估算值是 seq [nan; nan; nan; nan;…]。发生了什么情况？我们所面对的问题是**共线性**。在探讨这一问题的根源之前，我们先来看看说明问题根源的更简单的例子。回忆我们的第一个模型，特征的定义如下：

$$Y=\theta_0+\theta_1 \times obs.Instant$$

现在想象一下，我们修改特征，用如下的规格代替：

$$Y=\theta_0+\theta_1 \times obs.Instant+\theta_2 \times obs.Instant$$

这当然是愚蠢的做法，但是我们可以花点时间看看，在这种情况下，会发生什么？这里会碰到一个问题，θ_1 和 θ_2 有无限个组合，假定初始模型中 θ_1 的最优值为 10，则 θ_1 和 θ_2 相加为 10 的任何组合都工作得一样好，我们没有办法决定其优劣。这本质上就是该算法所面对的同一个问题：无法找到独一无二的最优 θ 值，因而遭到惨败。

那么，这里为什么会遭遇共线性问题呢？仔细思考一下，如果任何特征都可以表现为其他特征的线性组合，就会碰到这个问题。任何日期都是每周 7 天之一，所以星期一+星期二+……+星期日=1.0。但是 1.0 也是我们的常数项值——映射到 θ_0。我们该怎么办？

可以采取两种措施。首先，可以从列表中删除一天。这样，我们实际上将这天作为“参考日”，赋予其余 6 天的 θ 将捕捉和该日相比的微分增益或者损耗。另一种方法是从方程中删除常数项，那样，我们最终可以得到 7 天的直接估算。

我们将删除星期日，将其作为参考点。现在，估算很顺利，甚至可以得到小的质量改进：

```
let featurizer2 (obs:Obs) =
    [   1.
        obs.Instant |> float
        obs.Hum |> float
        obs.Temp |> float
        obs.Windspeed |> float
        (if obs.Weekday = 1 then 1.0 else 0.0)
        (if obs.Weekday = 2 then 1.0 else 0.0)
        (if obs.Weekday = 3 then 1.0 else 0.0)
        (if obs.Weekday = 4 then 1.0 else 0.0)
        (if obs.Weekday = 5 then 1.0 else 0.0)
```

```
        (if obs.Weekday = 6 then 1.0 else 0.0)
    ]

let (theta2,model2) = model featurizer2 training

evaluate model2 training |> printfn "Training: %.0f"
evaluate model2 validation |> printfn "Validation: %.0f"

>
Training: 721
Validation: 737
```

稍做检查，为对应周日匹配最后 6 个相关系数，就会发现周六似乎是自行车最受欢迎的一天，而周日人们会将时间用在其他方面。

我们不再将更多的分类特征加入模型了，但是你尽可以尝试，看看能够从中得到多大的改进！在此，我们强调两个要点：首先可以将各个类别划分为单独特征并使用指标变量（1 表示该类别，0 表示其他类别）表示其活跃度，这样就可以在工作量最小的情况下，于回归模型中使用分类变量；其次，我们讨论了共线性及其对模型的严重破坏，以及处理这一问题的方法。

4.6.2 非线性特征

我们的模型中已经包含了所有特征，还能够尝试其他方法改进预测吗？

为了解答这个问题，我们首先再观察一下数据，绘制自行车使用量与温度的对比图表（参见图 4-12）：

```
Chart.Point [ for obs in data -> obs.Temp, obs.Cnt ]
```

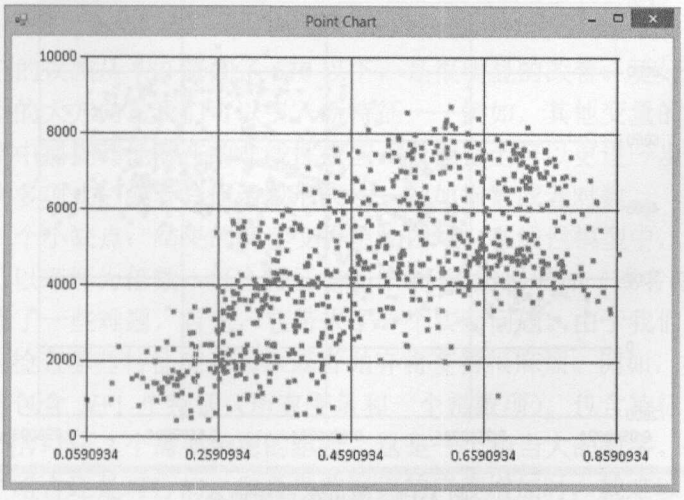


图 4-12 自行车使用量与温度对比的散点图

数据相当凌乱，但是总体而言，使用量随温度的上升而上升，到达某一点之后开始下降。这并非不合理的现象：在严寒和酷暑中，自行车都会失去吸引力。遗憾的是，当前的模型没有很好地捕捉这一因素。所有的影响都以线性函数表现，也就是说，它们是以“θ×特征”的形式出现的。如果 θ 为正数，在其他条件不变的情况下，特征值越高，预测值就越高；相反，如果 θ 为负数，特征值越高，预测值总是越低。

这并不能阻止我们捕捉温度的影响。我们预计，使用量在温度很低或者很高时都会下降。那么，该怎么做？

如果线性关系不够用，可以使用其他方法，例如高次多项式。最简单的方法之一是在组合中加入一个二次项，原来的温度影响模型如下：

$$obs.Cnt = \theta_0 + \theta_1 \times obs.Temp$$

现在我们改用如下模型：

$$obs.Cnt = \theta_0 + \theta_1 \times obs.Temp + \theta_2 \times obs.Temp^2$$

我们暂时忽略其他所有特征，尝试这一思路，以说明上述方程生成何种曲线（见图 4-13）：

```
let squareTempFeaturizer (obs:Obs) =
    [    1.
         obs.Temp |> float
         obs.Temp * obs.Temp |> float ]

let (_,squareTempModel) = model squareTempFeaturizer data

Chart.Combine [
    Chart.Point [ for obs in data -> obs.Temp, obs.Cnt ]
    Chart.Point [ for obs in data -> obs.Temp, squareTempModel obs ] ]
```

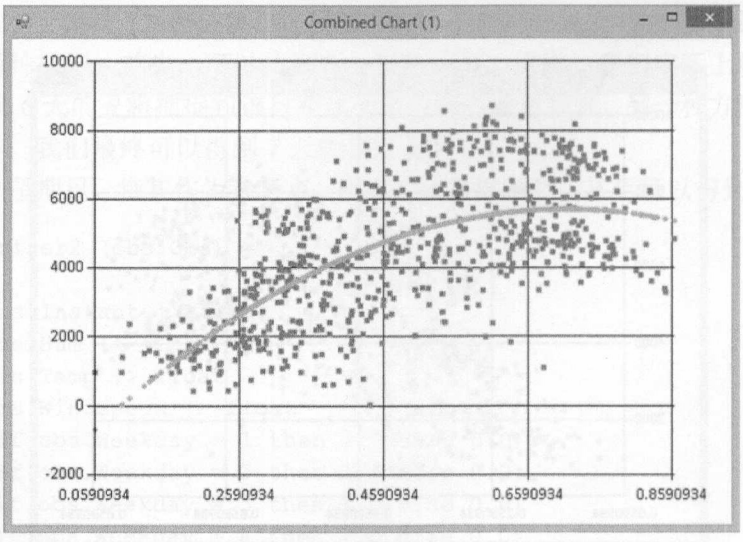


图 4-13　温度的二次拟合

我们得到的不是一条直线，而是一条弯折角度较小的曲线。这条曲线达到最大值之后开始下降。这种方程生成的一类曲线称作抛物线，总是遵循相同的一般形状（在最小值或者最大值的两侧对称），由于这种特性，在捕捉某一特征以特定值为峰值的模型时，它自然地成为候选。

现在，让我们将新特征加入模型中，看看能否带来改进：

```
let featurizer3 (obs:Obs) =
    [    1.
         obs.Instant |> float
         obs.Hum |> float
         obs.Temp |> float
         obs.Windspeed |> float
         obs.Temp * obs.Temp |> float
         (if obs.Weekday = 1 then 1.0 else 0.0)
         (if obs.Weekday = 2 then 1.0 else 0.0)
         (if obs.Weekday = 3 then 1.0 else 0.0)
         (if obs.Weekday = 4 then 1.0 else 0.0)
         (if obs.Weekday = 5 then 1.0 else 0.0)
         (if obs.Weekday = 6 then 1.0 else 0.0)
    ]

let (theta3,model3) = model featurizer3 training

evaluate model3 training |> printfn "Training: %.0f"
evaluate model3 validation |> printfn "Validation: %.0f"
>
Training: 668
Validation: 645
```

太好了！我们的误差从 750 降到了 670 以下，是很明显的改善。更好的是，我们现在有了进一步改进模型的大方向。我们可以引入新特征——例如，其他变量的平方或者两个变量的乘积——在模型中捕捉其他潜在的非线性效应。我们甚至可以更进一步，扩展到更高次的多项式，比如 3 次多项式，或者其他函数形式——例如指数或者对数。

这种方法有一个小缺点：结果的解读变得更加困难。在线性模型中，每个 θ 系数的含义很容易理解，它可以理解为倍数：每个特征增加 1.0，输出将增加（或者减少）θ。

这种方法造成了一些难题，首先，它带来了一个实际问题。由于我们扩大了所考虑的潜在特征数量，人工检查哪些特征应该加入或者抛弃将变得很麻烦。例如，如果有 N 个可用的变量，线性模型将包含 N+1 个特征（所有变量和一个常数项）。包含特征及其平方之间的相互作用将增加 $N \times (N+1) / 2$ 个需要考虑的组合，这是一个相当大的数字。对于中等数量的变量，人工检查每个组合还是可行的，但是当数据集的广度增加时，就难以为继了。

4.6.3　正规化

当然，更大的问题是我们前面讨论的要点。如果可以将所有特征一股脑儿地投入到模型中，那就好了。但是，正如前面所见，盲目地这么做只是机械地提高在训练集上的质量，但是不一定能够产生稳定的预测模型。这一问题被称作**过度拟合**：搜索算法使用所有可用的特征以拟合数据集，最终，在该任务上做得过分好了，可能捕捉到训练数据集中无法推广的特性。

那么，如何避免这一问题？常用的方法之一称为**正规化**（**Regularization**），思路如下：过度拟合的症状之一是，在尝试用特征拟合训练集时，算法开始产生某些“疯狂”的 θ 系数值，而在“合理”的模型中，系数应该保持在某一范围内，使得输入的变化不会造成疯狂的波动。

我们不介绍完整的正规化示例，而是简单地概述通常的做法。

如果目标是避免每个 θ 参数的波动不会太大，可能的做法之一是引入一个补偿函数（亦称作惩罚函数），使高的单独值在代价上高于小的单独值。例如，我们可以在代价函数中引入如下补偿函数：

$$Penalty(\theta)=l\times[\theta_1^2+\cdots+\theta_N^2]$$

为什么上述补偿函数是好的候选？正如前面讨论原始代价函数时所讨论的，因为 θ 值与 0 的距离越来越远时，平方函数的增长越快，较大的 θ 值得到的补偿越多。因此，默认情况下，上述补偿函数有利于较为平衡的值，而不利于差别很大的值。例如，θ 值[1.0; 1.0]的代价为 2，而[0.0; 2.0]的代价为 4。

将补偿函数与原来的代价函数相结合，可以用如下公式代替原来的代价函数：

$$cont(\theta)=\frac{1}{N}[(Y_1-\theta\times X_1)^2+\cdots+(Y_N-\theta\times X_N)^2]+l\times[\theta_1^2+\cdots+\theta_N^2]$$

这一公式的解读相当简单。模型的代价结合了两个元素：拟合误差有多大，以及参数有多小。换言之，现在增大一个参数 θ 改进拟合是有代价的，算法“偏爱”使用较为平衡参数的拟合改善。

在观察上述公式在实际算法中的应用之前，需要考虑几个重要的技术问题。

第一个需要考虑的问题是，为了保证上述公式有效，每个特征都必须使用可比较的标度。这是因为我们引入的补偿项假定每个系数 θ 中的类似变化可以相互比较。对于用补偿项实现的目的，可以描述为“当通过增大特征系数可以改善拟合质量时，偏向于增大具有最小系数的特征。”只要特征本身是可以比较的，这就是很好的标准。例如，在我们的例子中，obs.Instant 的范围为 1～731，而 obs.Windspeed 的范围为 0～0.5。将 Instant 的系数增大 1，将造成 1～731 之间的输出变化，而 Windspeed 上相同变化的最大影响只有 0.5。

处理这个问题的方法称作**规格化**。规格化重新标度模型中的所有特征，使其可以相互比较。有几种方法可以完成规格化，最简单地是应用如下转换，将每个特征按比例转换成 0～1 的数：$normalized=(x-x_{\min})\,/\,(x_{\max}-x_{\min})$。

第二个值得一提的要点是，在正规化补偿项中忽略了 θ_0。这并不是打字错误，首先，我们很难找到规格化常数项的方法，因为该特征的每个观测值都为 1。其次，θ_0 起的是偏移值的作用：对于其他规格化特征，θ_0 设定了模型的基准，可以是任何值。

第三个要点（也是最后一点）是补偿项之前的参数 λ，这个参数可以解释为权重，它驱动正规化补偿函数在代价函数中的价值。该参数需要人工调整，以确定哪一种正规化级别最有效。这是一种习以为常的过程：在验证集上尝试各种补偿水平，找出最有效的 λ 值，也就是具有最小误差的值。

假定数据集已经可靠地规格化，我们可以使用这个新公式轻松地更新原来的梯度下降算法，更新的规则如下：

$$\theta_1 \leftarrow \theta_1 - 2a \times \frac{1}{N}$$

$$\times[obs_1.Instant \times (\theta_0+\theta_1 \times obs_1.Instant - obs_1.Cnt)+\cdots$$
$$+obs_N.Instant \times (\theta_0+\theta_1 \times obs_N.Instant - obs_N.Cnt)\,]-l \times \theta_1$$

4.7 我们学到了什么?

面对现实吧，这是密集的一章，我们讨论了许多思路，花费了相当多的时间，以数学的方法讨论很抽象的问题。让我们看看，能否重新组合和总结本章的关键知识。

4.7.1 用梯度下降最大限度地减小代价

本章讨论的最重要的技术适用于机器学习的多种领域，而不仅仅是回归。我们介绍了梯度下降算法，这是一种按照梯度（从当前状态最陡峭的下降方向）迭代调整参数，确定使函数值最小的参数的通用方法。

梯度下降初看似乎有些可怕，这是因为其中大量地使用了微积分，但是一旦克服了最初的障碍，该算法就变得相当简单了。更有趣的是，它适用于广泛的场合。本质上，只要函数存在最小值、可导（我们需要一个导数!）且不太病态，就适合于这种方法。因此，它是解决大部分机器学习问题的出色常用工具：如果试图找到一个最适合于某个数据集的模型，打算最小化某种形式的距离或者代价函数，梯度下降就有可能帮助到你。

我们还了解了该算法的两个变种——随机和批量梯度下降，它们各有自己的优势和局限性。随机梯度下降方法每次用一个观测值更新参数，不需要一次操作整个数据集，因此对于在线学习很方便。如果数据集随时间推移而扩展，随时接收新观测值，就可以用刚收到的最新数据点更新预测，并不断地逐步学习。缺点是，正如在例子中所看到的，这种方法相对较慢，稳定性也可能较差：虽然算法总体上保持正确的方向，但是每个单独步骤不能保证改善。相比之下，批量梯度下降方法一次操作整个数据集，每一步都产生改善，缺点是每一步都需要更多的计算，而且，使用整个数据集可能不现实，特别是在数据集很大的情况下。

4.7.2　用回归方法预测数字

本章的主要目标是用目前所拥有的当日数据，预测特定日期自行车共享服务的用户数。这一模型被称作回归模型。和目标是预测观测值所属类别的分类模型不同，我们试图找出一个函数，组合各个特征以预测某个数字值，并尽可能拟合以接近真实数据。

从可能的最简模型开始，通过数据拟合一条直线，逐步包含更多特征以建立更为复杂的模型，说明了几种简单的技术。特别是，我们阐述了如何超越连续特征，将分类特征分解为几个特征，每个特征以 1 或者 0 表明活跃状态，将其加入模型中。

我们还了解了如何加入非线性项（如二次或更高次多项式），用于建立输入和输出之间无法用线性模型捕捉的复杂关系。

在模型中添加更复杂的特征是双刃剑，更好的预测需要付出代价。首先，这会造成特征数量的突增，管理所要包含的特征本身也成为问题。其次，还会引入更深层的问题：不管特征是否与手上的问题有关，添加任何特征都将机械地改善模型与训练集的拟合。这一问题被称作过度拟合：特征越多，算法就有更大的“自由度”，以创造性、扭曲的方式使用它们，更好的拟合并不一定能够转换为更好的预测。

我们讨论了使用交叉验证和正规化，作为缓解上述问题的技术方案。然而，作为最后的提醒，考虑固有的矛盾可能很有用。模型的实用性可能来自于两个截然不同的原因：理解某个现象，或者做出预测。添加更多的数据通常有助于做出更好的预测，但是也会引入复杂性。相反，简单的模型精确度较低，但是有助于理解变量的相互作用，可以验证某些事实是否有意义。在构建模型之前，先问问自己所偏爱的是精确度很高的模糊黑盒模型，还是精确度较低、但是容易向同事解释的模型。这有助于指导你的决策！

实用链接

- FSharp.Charting 是一个稳定的程序库，可以方便地用于在 FSI 中探索数据时创建基本图表：http://fslab.org/FSharp.Charting/。
- Math.NET 是一个开源库，涵盖广泛的数学和数值分析工具，特别是包含必不可少的线性代数算法：http://www.mathdotnet.com/。
- 如果需要从算法中得到更高的性能（即使这些算法需要较复杂的实现），quantelea 提供了非常有趣的.NET-GPGPU 库，对 F#特别友好：http://www.quantalea.net/。
- DiffSharp 在本书编写期间仍在发展，但它是一个有前途的自动微分库，可以显著简化梯度计算：http://gbaydin.github.io/DiffSharp/。

第 5 章

你不是独一无二的雪花

用聚类分析和主成分分析发现模式

有时候，你的机器学习始于非常特殊的问题。你能否断定一封入站邮件是不是垃圾邮件？你预测销售量的精确度如何？你已经吹响了冲锋号：从一个问题开始，找出可用的数据，建立回答问题的最佳模型。

但是，生活往往不是这样轮廓鲜明的，你的工作可能从更加模糊的问题开始。你所能拥有的是数据，但是还没有任何清晰的问题。可能有人问你："可以告诉我关于这些数据的趣事吗？"你可能想对某个数据集进行某些分析，但是希望首先理解曾经使用过的方法。数据集可能很大，有许多特征，如果特征之间没有明显的关系，就难以掌握。人工数据探索是痛苦而乏味的过程——如果我们可以将一些工作交给机器，让它理解数据中存在的模式，那就再好不过了。

这就是**无监督学习**的领域。在有监督学习方法（如分类或者回归）中，我们从一个问题和带有标签的数据（即带有问题答案的例子）开始，由此学习一个拟合数据的模型。相比之下，在无监督学习中，我们只拥有数据，希望计算机帮助我们找出数据中的某种结构。从人类的意义上，该结构可能是"有趣"的，可用于以有意义的方式组织单独元素。

在本章中，我们将完成如下工作：

- 介绍两种经典的数据分析技术——k-均值聚类分析和主成分分析，这两种方法的目标都是从数据集中自动提取信息，提供更简单实用的表现形式。我们将对来自 StackOverflow 网站的历史数据应用这些技术，观察出现的模式是否有趣。
- 概述相同的通用思路——发现数据中的模式——如何根据用户的过去行为和在其他用户中观测到的模式之间的相似度，对其做出建议。

■ 注意：本章中的一些较为复杂的图表已经包含在源代码包中，如果必要的话可以用更大的格式查看。

5.1　发现数据中的模式

略做思索，你就将意识到这不是件容易的事。如何定义关于数据的“趣事”？“有趣”没有清晰的指标，那么，我们究竟要让计算机做什么？

数据可能在很多方面上“有趣”。从另一面看，“无趣”的数据就是没有突出特征的数据。从这个意义上讲，图 5-1 显示的就是无趣的数据。

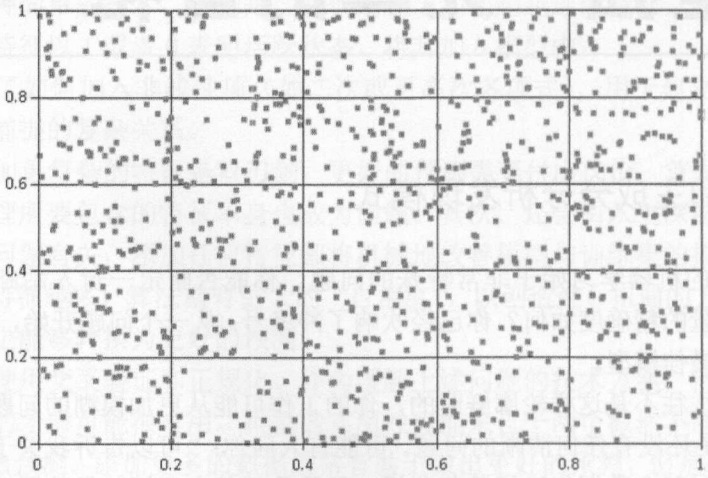


图 5-1　没有模式的数据

相比之下，图 5-2 显示了“某些趣事”。数据似乎有些组织，而不是均匀、同质的分布。观测值分为两组，围绕两个中心随机分布，这类模式称作“**聚类**”（Cluster）。聚类分析是自动识别数据集中此类结构的过程。

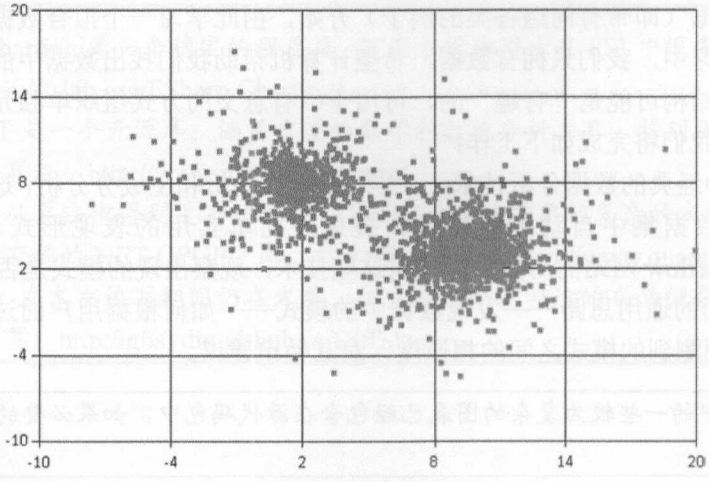


图 5-2　组织为两个聚类的数据

但是，套用列夫•托尔斯泰的名言，“无趣的数据集都是一样的，但是有趣的数据集各有自己的有趣之处。”聚类仅仅是数据集展现出“趣味”的许多方式之一。例如，考虑图 5-3，数据明显展示出有趣的模式，观测值大致沿着一条直线排列，但是没有形成聚类。

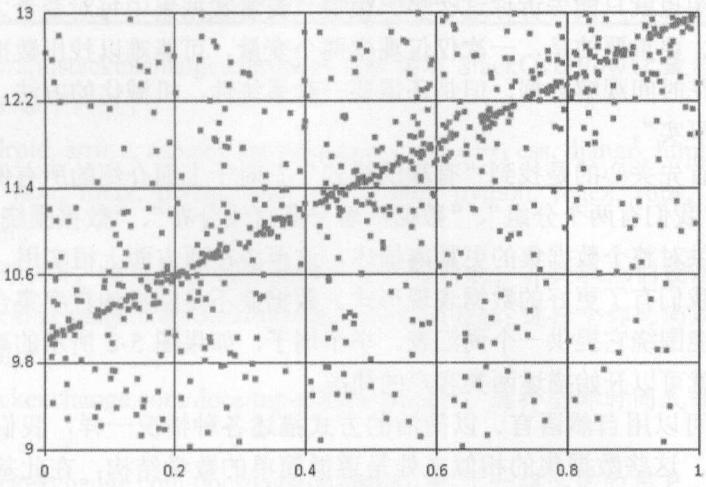


图 5-3 线性模式

聚类和线性结构是数据集所展现出来的两种有趣模式，但是，数据模式实际上是无限的；例如，图 5-4 中的数据组织为多个环形。有些像聚类（可以区分为 3 个组）也有些类似于图 5-3（数据遵循曲线）——但是也有明显的不同之处。

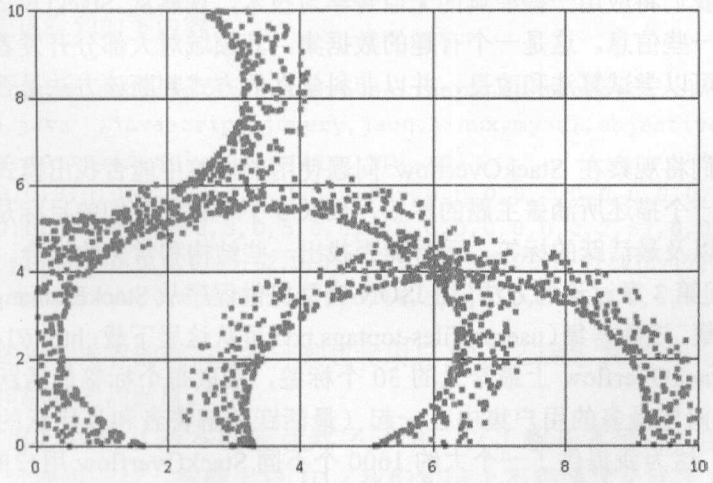


图 5-4 复杂模式

结构可以许多形式出现，因此，没有一种方法适用于所有情况。每个数据集都是独一无二的，数据挖掘从某种意义上说是神秘的艺术。找出有效的模式往往需要尝试不同的算法，并操纵数据、将其转换为更合适的特征。

人类非常擅长发现模式。你甚至可以不假思索地说出我刚才介绍的例图中的模式。在这个框架下，图表是你的朋友：制作一个特征与另一个特征相比较的散点图，是发现数据结构简单、高效的手段。

同时，散点图也就只能告诉你这么多。很快，查看数据集中每对变量之间的相互关系就变得令人痛苦了。更重要的是，一次仅仅观察两个变量，可能难以找出数据中存在的模式。所以，虽然应该花时间观察数据，但是还需要一种系统性、机械化的方法，筛选数据集，找出潜在的“有趣事实”。

为什么我们首先关心的是找到“有趣的模式”？对于上面介绍的所有例子，可以用一句话描述数据集：“我们有两个分组”、“数据沿着一条直线分布”、“数据围绕 3 个环形组织”。换言之，可以提供对整个数据集的更紧凑描述。这至少在两方面上很实用。

首先，现在我们有了更好的数据表现形式。数据集不是随机的数字集合，而是以有意义的方式描述，可能围绕它提供一个词汇表。举个例子，如果图 5-2 所示的聚类图表中数据点代表客户，我们就可以开始描述两类客户的简况。

其次，正如可以用自然语言、以简洁的方式描述各种情况一样，我们还可以提供简洁的数学表现形式。这些数据集的相似之处是遵循简单的数学结构，在此基础上还有少数统计噪声。

5.2　我们所面临的挑战：理解 StackOverflow 上的主题

在本章中，我们将应用一些经典的无监督学习技术，观察从 StackOverflow 使用情况数据中能否挖掘出一些信息。这是一个有趣的数据集，其领域对大部分开发者来说都应该很熟悉。因此，我们可以尝试算法和改良，并以非科学性的方式判断该方法是否产生了有意义的结果。

特别是，我们将观察在 StackOverflow 问题使用的标签中能否找出模式。StackOverflow 的每个问题都有一个描述所涵盖主题的标签，最长 5 个单词。我们的目标是通过观察一段时间的用户和问题以及最活跃的标签，看看能否找出一些结构和常见的组合。

为此，我使用第 3 章讨论的方法（用 JSON 类型提供程序从 StackExchange API 收集数据）创建了一个数据集。该数据集（userprofiles-toptags.txt）可从这里下载：http://1drv.ms/1ARuLg9。我找出了当时 StackOverflow 上最常见的 30 个标签，读取每个标签最活跃的用户，将提供答案最多和提出问题最多的用户集中在一起（最活跃的解答者和最活跃的提问者，全部时间和当月数据）。这为我提供了一个大约 1600 个不同 StackOverflow 用户的列表。

然后，对每个单独的用户，读取他们 2015 年中积极参与的 100 个标签，以及对应的活跃度，仅保留属于 30 个主要标签的数据。最后，在最终数据集中重新排列数据，组织为 30 列（每个标签一列）、1600 行（每行一个用户），每行和每列都包含用户 2015 年之内活跃于该标签的次数。注意，因为我只读取了用户前 100 个标签的数据，所以每个标签的信息都不

完整。例如，如果某个特定用户活跃于 XML，但是这个标签在他最活跃的标签中只排在第 101 位，那么我们的记录就是 0。

数据集是如何创建的?

我们用 https://api.stackexchange.com/docs/tags 读取 StackOverflow 上最流行的 30 个标签，在创建时，产生了如下列表：

.net, ajax, android, arrays, asp.net, asp.net-mvc, c, c#, c++, css, django, html, ios, iphone, java, javascript, jquery, json, linux, mysql, objective-c, php, python, regex, ruby, ruby-on-rails, sql, sql-server, wpf, xml。

我们使用如下方法，按照标签创建最活跃用户的列表，并提取他们最活跃的标签。

https://api.stackexchange.com/docs/top-answerers-on-tags：选择全部时间或者最近的一个月提供最多答案的用户。

https://api.stackexchange.com/docs/top-askers-on-tags：选择全部时间或者最近的一个月提出最多问题的用户。

https://api.stackexchange.com/docs/tags-on-users：对于一组选定的用户，返回最活跃标签的活跃度。

鉴于任务的探索特性，我们将再次在 F#脚本环境中工作。我们创建一个新解决方案，包含 F#库项目 Unsupervised，为了方便起见，将数据文件加入解决方案。数据集的形式是一个文本文件——userprofiles-toptags.txt，可以从如下链接下载：http://1drv.ms/1M727fP。从 Visual Studio 中查看，应该看到如下内容：

```
    UserID,.net,ajax,android,arrays,asp.net,asp.net-mvc,c,c#,c++,css,django,
html,ios,iphone,java  ,javascript,jquery,json,linux,mysql,objective-c,php,python,
regex,ruby,ruby-on-rails,sql,sqlserver, wpf,xml
    1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    1000343,0,0,0,0,0,0,0,0,0,3,0,5,0,0,0,0,0,0,0,0,0,0,2,52,0,0,0,0,0,0
    100297,0,0,0,26,0,0,0,0,0,0,99,62,0,0,0,29,0,182,0,26,0,0,4478,172,0,0,32,0,0,27
    100342,0,0,0,0,0,0,1,0,0,0,0,0,3,0,0,0,16,5,0,0,0,0,0,0,0,1,7,0,0,0,0
```

第一行是描述每列内容的表头。第一列包含用户 ID，之后是 30 个用逗号分隔的列，每列包含特定用户和标签的活跃度。

研究算法之前，我们从基本的统计数字开始，以了解基本情况。我们将首先打开 Script.fsx 文件，读取每一行，删除用户 ID（我们实际上不需要这个字段），将每行解析为一个浮点数组。我们也可以将数值保存为整数，但是因为可能进行平均等运算，直接转换为容易处理的类型可能更好。我们还将把表头保存在一个单独的数组中，以便在以后将列映射为正确的标签名。

程序清单 5-1　将数据集读入内存

```
open System
open System.IO

let folder = __SOURCE_DIRECTORY__
let file = "userprofiles-toptags.txt"

let headers,observations =

    let raw =
        folder + "/" + file
        |> File.ReadAllLines

    // first row is headers, first col is user ID
    let headers = (raw.[0].Split ',').[1..]

    let observations =
        raw.[1..]
        |> Array.map (fun line -> (line.Split ',').[1..])
        |> Array.map (Array.map float)

    headers,observations
```

数据就绪之后，我们通过计算平均值、最小值和最大值等统计数字，看看每个变量的样子。我们将使用两个小技巧。Array.iteri 可以循环读取一个数组，捕捉当前索引，结果是，我们可以循环读取表头，并使用当前索引 *i* 从观测值中提取第 *i* 列。第二个技巧是，在 printfn 中使用高级格式化符，强制使用一致的列宽和小数点位数，使输出更容易辨认。举个例子，printfn "%4.2f" 12.34567;;使用了格式化符%4.2f，只保留小数点后面两位，并将该列填充到 4 个字符（除非原来的字符数足够）。

程序清单 5-2　基本数据集统计数字

```
printfn "%16s %8s %8s %8s" "Tag Name" "Avg" "Min" "Max"

headers
|> Array.iteri (fun i name ->
    let col = observations |> Array.map (fun obs -> obs.[i])
    let avg = col |> Array.average
    let min = col |> Array.min
    let max = col |> Array.max
    printfn "%16s %8.1f %8.1f %8.1f" name avg min max)
>
        Tag Name Avg Min Max
            .net 3.5 0.0 334.0
            ajax 1.7 0.0 285.0
```

```
        android 7.1 0.0 2376.0
         arrays 6.4 0.0 327.0
         asp.net 2.6 0.0 290.0
// snipped for brevity
```

简单地浏览输出，就可以发现几个突出的特征。首先，每个标签的最小值都为 0，这意味着每个标签都有无活跃用户的情况。其次，在每种情况下，平均值都很小，和最大值相比，它与 0 的距离更近。这说明对于每个标签，大量用户的活跃度都很低（甚至没有任何活动），只有少数人活跃度较高。考虑到我们收集了每个标签最活跃用户的使用数据，这并不完全出乎意料。每个标签的活跃度分为两组（非常活跃或者非常不活跃），没有中间状态，而不像经典的钟形分布——数值围绕平均值展开。

数字表格过于低级，无法从中识别出大的趋势，我们通过 NuGet 添加 FSharp.Charting，在一个柱状图上绘出每个变量（参见图 5-5），更好地感觉数据中的趋势。

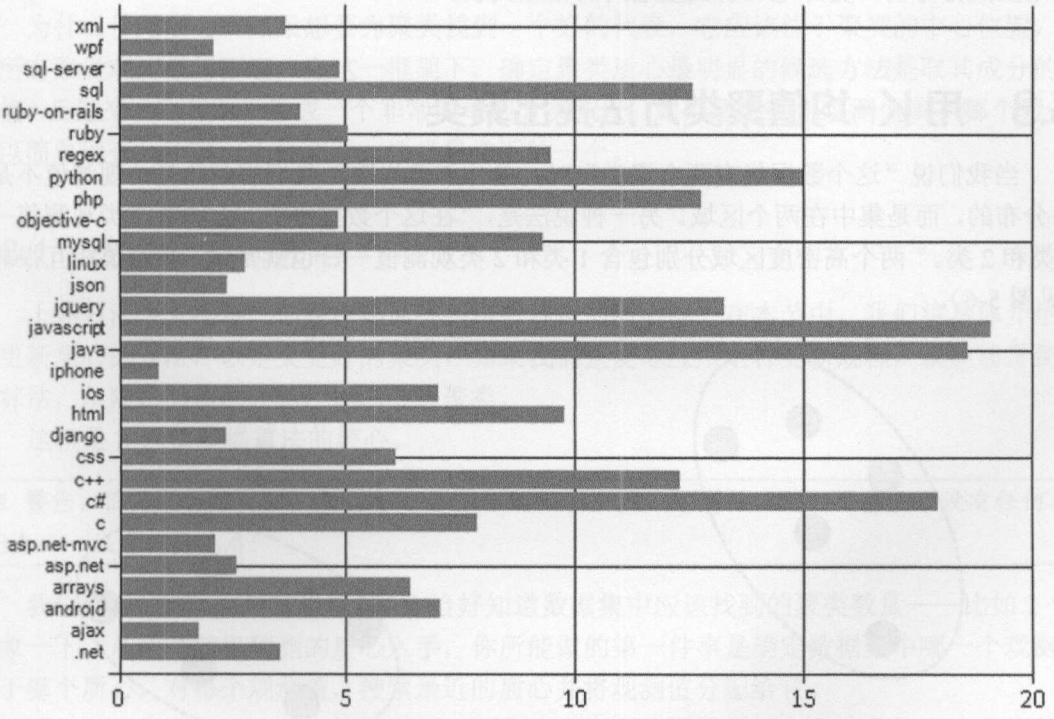


图 5-5 按照标签排列的平均使用量

程序清单 5-3 按照标签绘制平均使用量图表

```
#r @"..\packages\FSharp.Charting.0.90.9\lib\net40\FSharp.Charting.dll"
#load @"..\packages\FSharp.Charting.0.90.9\FSharp.Charting.fsx"
open FSharp.Charting
```

```
let labels = ChartTypes.LabelStyle(Interval=0.25)

headers
|> Seq.mapi (fun i name ->
    name,
    observations
    |> Seq.averageBy (fun obs -> obs.[i]))
|> Chart.Bar
|> fun chart -> chart.WithXAxis(LabelStyle=labels)
```

图表显示，有几个标签的使用量明显大于其他标签（Javascript、Java、 C#、 Python、jQuery、 PHP、SQL 和 C++），相比之下，其他标签的使用量小得多。这是有用的信息，但是有局限。我们希望用更全面的视角，帮助我们理解这些主题之间可能存在的关系，而不是隔离不同的事实。这是下一节的主题，首先使用 k-均值聚类识别有类似行为的用户组，然后使用主成分分析，更好地理解变量之间的相互关联。

5.3　用 K-均值聚类方法找出聚类

当我们说“这个数据集有两个聚类”时，是什么意思呢？我们想到的是：观测值不是均匀分布的，而是集中在两个区域。另一种说法是，“在这个数据集中，大约有两类观测值——1 类和 2 类。”两个高密度区域分别包含 1 类和 2 类观测值——也就是说，观测值相互很靠近（见图 5-6）。

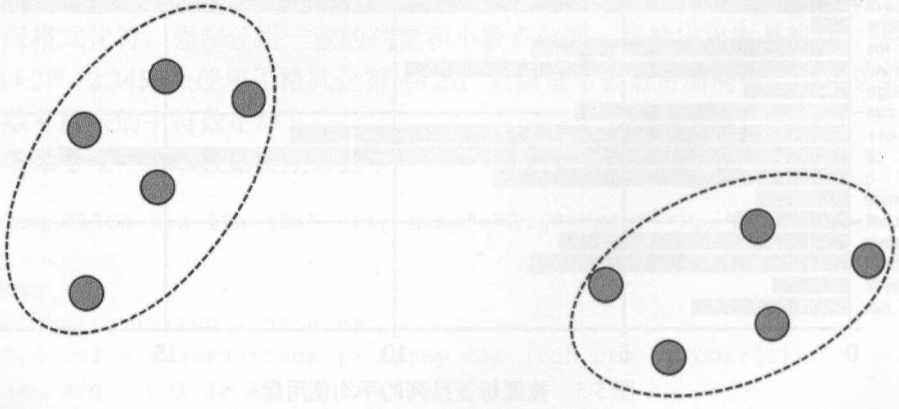


图 5-6　数据聚类

我们可以用稍微不同（可能更简单）的方式去看待这种现象。聚类中的所有观测值是理想化观测值的微小变化。这简化了许多问题：不需要检测聚类的边界，可以将聚类归纳为整个组的典型代表。这些“虚构代表”被称为**质心**（Centroid）。

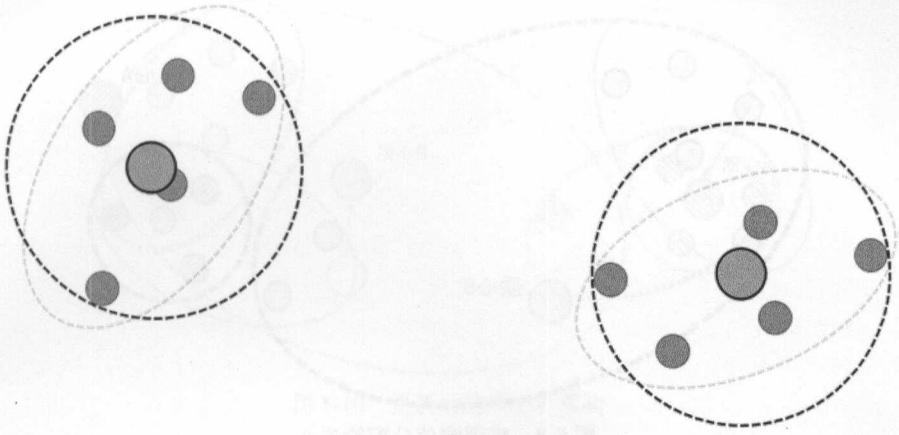


图 5-7 聚类和质心

为什么使用质心？如果想要为聚类找到一个好的代表，它应该位于聚类的中心位置，距离所有聚类观测值都不远。在这一框架下，确定聚类质心最明显的候选方法是取其成分的平均值。反过来，这有助于形成一个非常简单的分类规则：如果想要确定观测值属于哪个聚类，可以简单地查找所有可用的质心，选择最靠近的一个。

5.3.1 改进聚类和质心

上述方法令人满意，但是，如何找到这些神奇的质心呢？在本节中，我们将具体介绍一种更新质心的方法，以定义更好的聚类。如果我们重复地应用这种更新规程，就能够得到一个算法，从随机的质心开始，逐渐改进聚类。

这就是 **k-均值聚类算法**的核心。

■ **警告**：因为名称很类似，人们有时会将 k-均值和 k 最近邻算法混同。两者几乎没有任何共同点——注意不要混淆！

我们暂时假定，通过某种魔法，你恰好知道数据集中应该找到的聚类数量——比如 2 个。想象一下，从两个随机猜测的质心入手，你所能做的第一件事是确定数据集中哪一个观测值属于哪个质心。对每个观测值，搜索最近的质心并将观测值分配给它。

图 5-8 取得图 5-6 中的虚构数据集，两个“真实”的聚类用浅灰色边界标出。我们从两个随机的质心 A 和 B 开始，将每个数据点分配给最近的质心。颜色较深的边界大致表现了以当前质心为基础的聚类外观。

我们刚刚做的是定义候选聚类。分配给相同质心的所有观测值组成一个可能的聚类。如果当前聚类质心是“好的质心”，它应该位于聚类的中央。我们调整质心位置（见图 5-9），将其移向聚类的平均位置——中心。

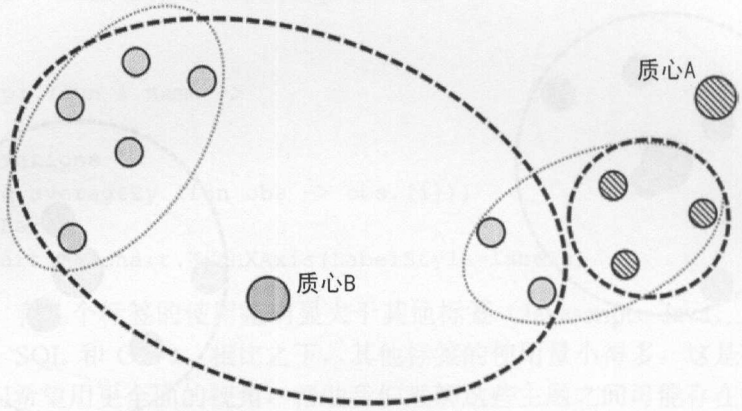


图 5-8　将观测值分配给质心

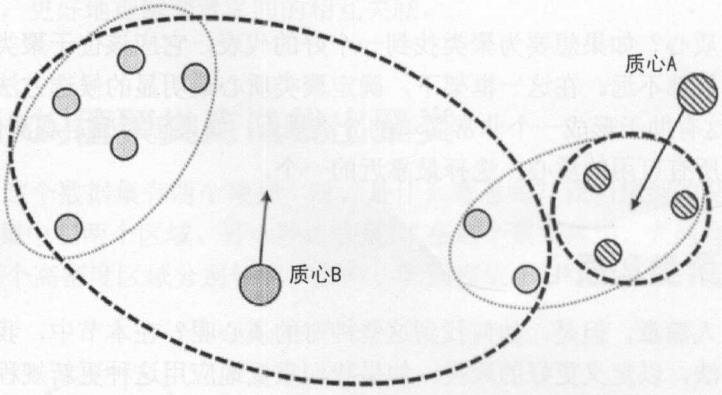


图 5-9　调整质心

与此同时，注意发生的两件趣事。将质心移向聚类中的平均位置，意味着被吸引到高密度区域，这个小区域中集中了更多重量，作为“吸引力来源”。举个例子，考虑图 5-10 中质心 B 所发生的情况，它很自然地移向左侧的“真实”聚类。在移动中，它远离边缘被隔离的观测值，例如右侧的两个观测值。这将从聚类中去除没有特性的元素，使数据点有可能被另一个质心吞噬。

更新进行之后，我们可以重复这一规程，重新计算每个观测值分配的质心，以及更新后的质心位置。

应该在何时停止更新？注意，如果质心位置更新前后，同一聚类分配的是相同的观测值，质心就不会再移动。每个质心已经处于聚类的平均位置，更新不会改变其位置。

这本质上就是 k-均值聚类算法的工作方式：将数据集分为 k 个聚类，选择 k 个随机的质心，逐渐更新其位置，直到达到稳定。这一解释应该足以让读者理解，为什么这种方法有效，为什么可以得到位于高密度聚类中的稳定质心。这方面的说明到此为止，我们不再尝试证明该方法能否真正奏效。

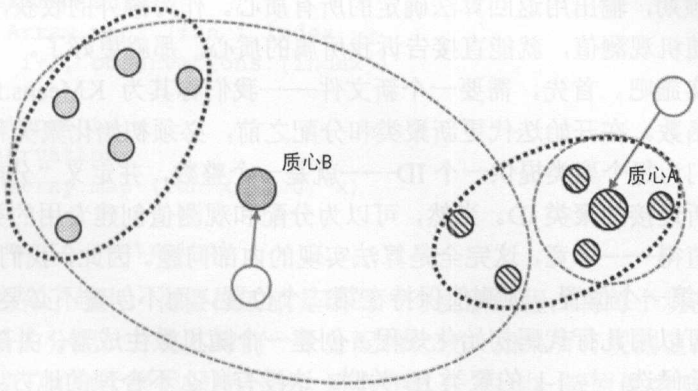


图 5-10 更新后的质心和聚类

该算法可以视为定点方法的应用。更新规程假定稳定的质心存在，试图逐步消除某个质心定义的聚类之间以及聚类定义的质心之间的差异。我们还有一些重要的问题没有回答，比如，如何求出 k 值（聚类数量）——但是首先，我们继续着手在 F#中实现上述算法，假定已经知道准确的 k 值，看看这会将我们带向何方。

5.3.2 实施 K-均值聚类方法

从算法的描述可以得到几个概念：给定一组观测值，我们必须根据最短距离将它们分配给 k 个质心中的一个，还需要将一组观测值归纳为一个质心。目前，数据集没有什么特殊性，因此我们将尝试编写一个算法的通用版本，将其放入单独的模块中，以后将其加载到一个脚本中，用于处理手上的问题。

这为我们提供了一个稳定的出发点。从外部看，算法应该是这样的：必须提供一组观测值和聚类的数量 k。我们还需要距离的定义，以便将观测值分配给最近的质心；还需要一种方法，将统一聚类的观测值归纳为单一的聚合观测值——更新后的质心。下面是使用伪代码编写的实施算法大纲：

```
聚类化
    1. 初始化
        为每个观测值分配 ID 为 0 的初始质心
        选择 k 个不同的观测值，作为初始质心 1-k
    2. 搜索
        将每个观测值分配给最近的质心
        如果分配值没有变化，完成：我们已经有了稳定的质心
        否则
            聚集分配给相同聚类的观测值
            将观测值归纳为平均质心
            以新质心重复搜索过程
    3. 从初始质心开始搜索
```

我至少可以预期，输出用返回算法确定的所有质心。作为额外的收获，得到一个“分类器”函数，指定随机观测值，就能直接告诉我所属的质心，那就更好了。

让我们开始实施吧。首先，需要一个新文件——我们称其为 KMeans.fs——然后，开始编写一个聚类化函数。在开始迭代更新聚类和分配之前，必须初始化聚类和分配，并决定如何表示它们。我们为每个聚类提供一个 ID——就是一个整数，并定义“分配”（assignments）为一个观测值和所连接的聚类 ID。当然，可以为分配和观测值创建专用的类型，但是在此花费精力似乎并不值得——毕竟，这完全是算法实现的内部问题。因此，我们简单地使用元组，将聚类 ID 保存在第一个位置，观测值保持在第二个位置，而不创建不必要的类型。

然后，我们可以用几行代码初始化规程。创建一个随机数生成器，并随机选择 *k* 个观测值作为质心的初始候选，与 1-k 的聚类 ID 关联。这没有什么不合理的地方：选择现有观测值可以保证质心的位置不是完全不现实的。与此同时，这也意味着有选择“坏”的初始质心的危险。我们暂时忽略这个问题，放在以后讨论。

第一个问题是从数据集中随机选择 *k* 个不同的观测值，作为初始质心。我们简单地选择从 0 到数据集大小−1 范围内的随机数，并添加到一个集合（这将自动检测并消除重复），直到选中的观测值数为 *k*——我们需要的聚类数量。这一方法并不完美，特别是，如果试图选择很大的一组（和数据集的大小相比）不同索引，该方法的效率很低。但是，假如聚类的数量相比起来很小，这种方法就是可行的。

程序清单 5-4　选择 *k* 个观测值作为初始质心

```
namespace Unsupervised

module KMeans =

    let pickFrom size k =
        let rng = System.Random ()
        let rec pick (set:int Set) =
            let candidate = rng.Next(size)
            let set = set.Add candidate
            if set.Count = k then set
            else pick set
        pick Set.empty |> Set.toArray
```

至于分配，我们将赋予随机的起始质心 ID0——不匹配任何真实的质心 ID。如果所有质心 ID 都大于 1，意味着在第一遍搜索中，每个观测值都保证分配了不同的质心，算法至少完成了一个完整的更新循环。

程序清单 5-5　初始化聚类算法

```
let initialize observations k =

    let size = Array.length observations

    let centroids =
```

```
        pickFrom size k
        |> Array.mapi (fun i index ->
            i+1, observations.[index])

    let assignments =
        observations
        |> Array.map (fun x -> 0, x)

    (assignments,centroids)
```

现在我们已经有了起点，需要做的就是编写一个更新规程，将每个观测值分配给最近的质心，得到一个新的分配，并将其与前一个分配相比较。如果所有观测值都没有改变所属的聚类，工作就完成了（每个聚类保持不变，质心亦然），否则，按照聚类 ID 集合观测值，将其归纳为平均质心，继续更新。我们将在 clusterize 函数中定义一个递归函数 search，生成初始值，开始递归地更新质心。

程序清单 5-6　递归聚类更新

```
let clusterize distance centroidOf observations k =

    let rec search (assignments,centroids) =
        // Centroid ID of the closest centroid
        let classifier observation =
            centroids
            |> Array.minBy (fun (_,centroid) ->
                distance observation centroid)
        |> fst
    // Assign each observation to closest centroid
    let assignments' =
        assignments
        |> Array.map (fun (_,observation) ->
            let closestCentroidId = classifier observation
            (closestCentroidId,observation))
    // Check if any observation changed cluster
    let changed =
        (assignments,assignments')
        ||> Seq.zip
        |> Seq.exists (fun ((oldClusterID,_),(newClusterID,_)) ->
            not (oldClusterID = newClusterID))

    if changed
    then
        let centroids' =
            assignments'
            |> Seq.groupBy fst
            |> Seq.map (fun (clusterID, group) ->
                clusterID, group |> Seq.map snd |> centroidOf)
```

```
            |> Seq.toArray
        search (assignments',centroids')
    else centroids,classifier

// start searching from initial values
let initialValues = initialize observations k
search initialValues
```

这就行了：我们有了一个聚类算法！给定一个观测值数组和 k 值，只要提供距离的定义和想要使用的归纳函数，就应该能够得到聚类和将所有观测值分配给对应聚类的函数。

5.4 StackOverflow 标签的归类

我们回到脚本文件，看看对数据集进行聚类分析时发生了什么。我们需要几个要素，以便运行聚类算法：在脚本中加载和打开 KMeans 模块，观测值、距离和工作于观测值之上的归纳函数，以及 k 值——搜索的聚类数量。

5.4.1 运行聚类分析

我们使用欧几里得距离，将一组观测值归纳为其平均值。使用如下代码很容易做到这一点，代码包含在我们的脚本中：

程序清单 5-7　为聚类定义距离和归纳

```
#load "KMeans.fs"
open Unsupervised.KMeans

type Observation = float []

let features = headers.Length

let distance (obs1:Observation) (obs2:Observation) =
    (obs1, obs2)
    ||> Seq.map2 (fun u1 u2 -> pown (u1 - u2) 2)
    |> Seq.sum

let centroidOf (cluster:Observation seq) =
    Array.init features (fun f ->
        cluster
        |> Seq.averageBy (fun user -> user.[f]))
```

现在，我们已经为搜索聚类做好了准备。我们将淘汰没有使用任何标签的观测值（也就是用户在此期间在所选择的标签上完全没有任何活动），完全随机地选择 k=5——然后调用 clusterize 函数：

程序清单 5-8 数据集聚类分析

```
let observations1 =
    observations
    |> Array.map (Array.map float)
    |> Array.filter (fun x -> Array.sum x > 0.)

let (clusters1, classifier1) =
    let clustering = clusterize distance centroidOf
    let k = 5
    clustering observations1 k

>
val clusters1 : (int * Observation) [] =
 [|(1,
    [|2.47860262; 1.063755459; 9.765065502; 5.454148472; 2.333624454;
      2.293449782; 5.173799127; 10.98689956; 4.822707424; 5.772052402;
      1.693449782; 7.274235808; 9.717030568; 1.220087336; 23.49956332;
      10.12751092; 5.999126638; 2.011353712; 2.509170306; 7.932751092;
      6.620087336; 12.78777293; 7.658515284; 7.425327511; 6.794759825;
      5.561572052; 9.624454148; 4.525764192; 1.031441048; 4.483842795|]);
    // snipped for brevity
    (2,
    [|0.0; 0.0; 19.0; 17.0; 19.0; 0.0; 0.0; 69.0; 0.0; 0.0; 0.0; 0.0; 0.0;
      0.0; 68.0; 0.0; 0.0; 0.0; 0.0; 4159.0; 0.0; 821.0; 17.0; 56.0; 0.0; 0.0;
      7054.0; 1797.0; 0.0; 0.0|])|]
val classifier1 : (Observation -> int)
```

正如预期，我们得到了两个结果：5 个聚类组成的数组，每个聚类有一个 ID 和一组代表该聚类在每个标签上配置的数值。举个例子，在上述输出中，(2, [|0.0; 0.0; 19.0; // snipped // 0.0|]) 表示 ID 为 2 的聚类，该聚类中的普通用户从不使用.NET 或者 ajax 标签（我们的列表中的前两个标签），但是使用 Android 标签 19 次。注意，因为初始质心是随机选取的，很有可能在每次运行中得到不同结果（我们将很快回到这个主题）。

5.4.2 结果分析

面对现实，按照当前的形式，输出是很难诠释的。对此有所帮助的第一件事是映射每个聚类，并将标签名称与我们所拥有的数值匹配。做法如下：

```
clusters1
|> Seq.iter (fun (id,profile) ->
    printfn "CLUSTER %i" id
    profile
    |> Array.iteri (fun i value -> printfn "%16s %.1f" headers.[i] value))
```

```
>
CLUSTER 4
            .net 2.9
            ajax 1.1
         android 6.1
           arrays 5.0
         asp.net 2.4
         // snipped for brevity
```

这比开始好一些了，至少，我们可以开始浏览每个聚类，搜索某种模式。但是，信息仍然有点太多了——我们来看看使用图表是否有助于更清晰地可视化输出（见图 5-11）：

```
Chart.Combine [
    for (id,profile) in clusters1 ->
        profile
        |> Seq.mapi (fun i value -> headers.[i], value)
        |> Chart.Bar
    ]
|> fun chart -> chart.WithXAxis(LabelStyle=labels)
```

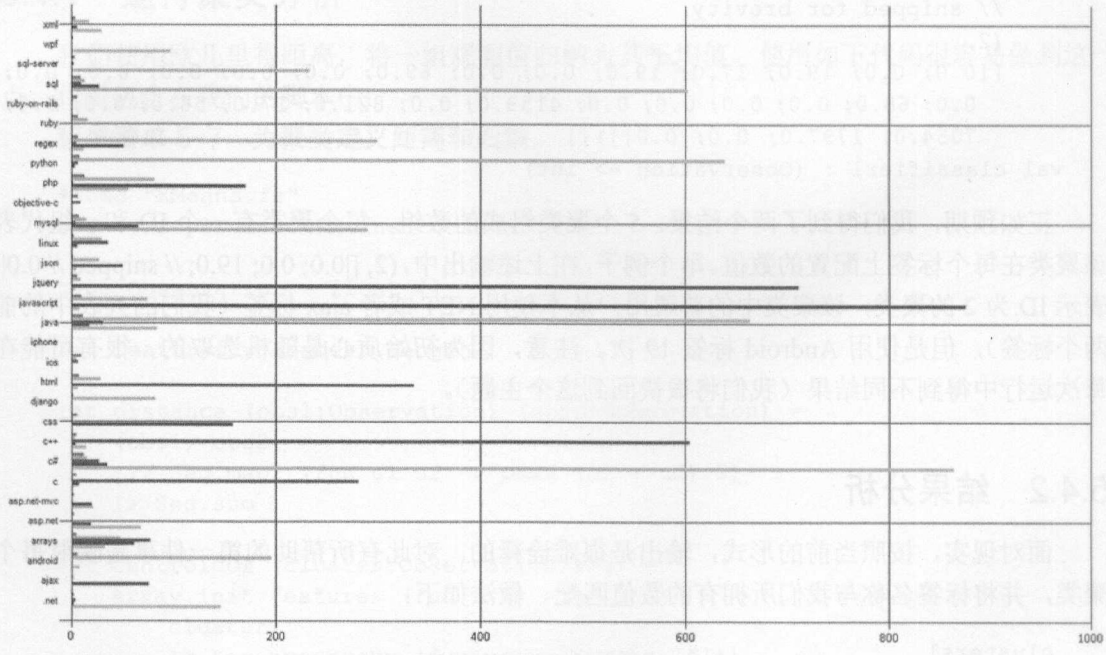


图 5-11　聚类简况（详见源代码包）

生成图表解析起来就容易多了。在本例中，我看到：

- 聚类 2 在 Java、Android、Python 和 Regex 上有很大的峰值。

- 聚类 4 在 Javascript、JQuery、html、css 和 PHP 上出现峰值。
- 聚类 5 在 C#、.NET、WPF，以及 SQL、SQL Server 和 MySQL 上取值较大。
- 聚类 3 的峰值出现在 C、C++、Linux 和 Arrays 上也有些线索。
- 聚类 1 没有明显的峰值。

这很有趣。一方面，聚类算法确实运行并生成了聚类，出现了一些有意义的关联。看到 Java 和 Android 出现在一起并不令人惊讶。JavaScript、JQuery 和 html 等 Web 客户端技术或者 SQL、MySQL 和 SQL Server 等数据库相关标签在同一类中也是合乎逻辑的。与此同时，也出现了一些奇怪的现象：Java、Python 和 Regex 真的相关吗？为什么这个奇怪的聚类中混合了.NET 和数据库开发？

更进一步之前，我们先看看另一部分信息。每个聚类的大小如何？

```
observations1
|> Seq.countBy (fun obs -> classifier1 obs)
|> Seq.iter (fun (clusterID, count) ->
    printfn "Cluster %i: %i elements" clusterID count)
>
Cluster 4: 1137 elements
Cluster 2: 22 elements
Cluster 1: 23 elements
Cluster 5: 20 elements
Cluster 3: 15 elements
```

这个情况就不怎么好了。我们有一个巨大的聚类，大约 93%的观测值都属于此类，另外还有 4 个聚类，每个包含的观测值都不到数据集的 2%。在一个完美的世界里，我们应该看到大小相近的聚类。这是怎么回事？

5.5 好的聚类和坏的聚类

我们后退一步，考虑一下我们的方法可能出现的错误，以及有希望的改进方法。首先，有一个明显的可能性：数据中可能完全没有聚类。从这个意义上讲，没有什么可做的——如果没有聚类，那就没有什么措施了。但是，考虑到最初的分析中看到了一些有意义的标签组，我们将假定自己没有失败，进一步探究其中的原因。

我们的算法至少有两个方面可能出错：寻找的聚类数量可能不合适，或者特征的标度差别太大，无法以有意义的方式比较。让我们更仔细地观察这两个问题，希望从中领悟出改善聚类的思路。

我们搜索的聚类数量是完全随机选定的——如果这个数字不对，预期的结果应该是怎样的呢？考虑图 5-12，图中展现出了 3 个相当清晰的聚类，想象一下要求寻找 2 个聚类时的情况。

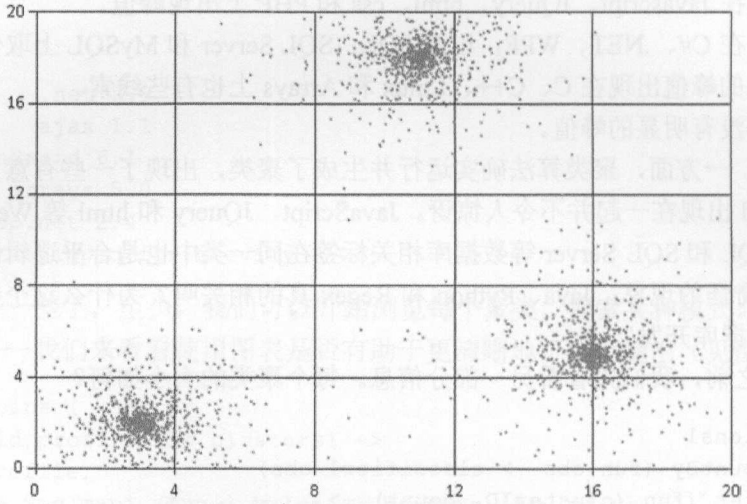


图 5-12　搜索的聚类数量不足

算法很有可能以 3 种方式中的 1 种确定 2 个聚类：其中一个聚类将是“正确的”，另一个聚类将包含剩下的两个聚类，如图 5-13 所示。

这很合理：如果我们只能选择两组，选择一个完美的分组，并由剩下的两个组的成员组成质量中等的一个组，是最好的结果。但是，由于初始质心是随机选择的，聚类几乎完全取决于开始选择的 3 个随机观测值。如果我们多次重复这一过程，就会看到 3 种不同结果，以不同的方式组合“真正”的聚类。在聚类的数量超过 3 个的情况下，应该会看到更多的组合。

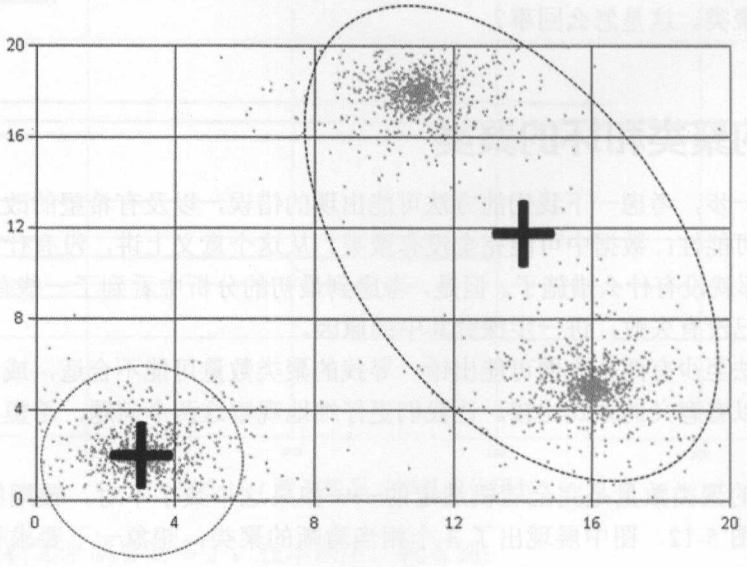


图 5-13　从 3 个聚类中搜索 2 个

这个问题和我们所观察到的似乎一致。如果多次运行聚类函数，可能观察到如下现象：两次运行的结果不同，但是同样的分组常常一起出现，只是可能和其他分组合在一起。举个例子，Java 和 Android 的组合经常出现，有时和 Python 在一起，有时不在一起。

如果要解决这个问题，必须将聚类数量增大到一个真正有效的值；也就是说，找出接近于实际聚类数量的 k 值。

我们将在下一小节中观察 k-均值聚类的另一种错误之后重新讨论这个问题。考虑图 5-14 中描述的情况。我们所拥有的是 2 个聚类，以{-2;0}和{2;0}为中心，但是在 X 轴上和 Y 轴上的延伸不同。X 值的范围主要在[-3;-1]和 [1;3]之间，而 Y 值延伸到−20 到 20 之间。

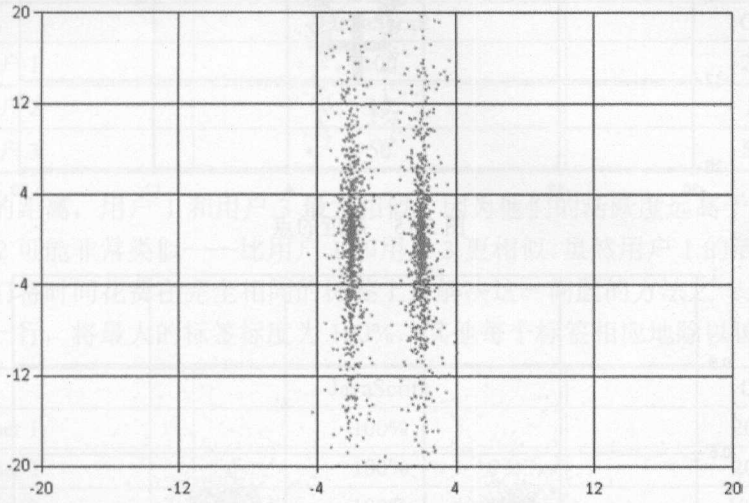


图 5-14 标度大不相同的特征

如果我们直接在原始数据上使用 k-均值方法，要求找出两个聚类，会发生什么？同样，这里有一个随机因素，取决于我们所选择的初始质心，但是，结果似乎很合理：我们很容易找到两个聚类，一个以{0；10}为中心，另一个以{0，−10}为中心。为什么？

问题在于，两个特征的标度大不相同，因此，X 轴（第一个特征）上的任何差异在 Y 轴上的分布范围面前都相形见绌。换言之，在原始数据上计算距离所得出的两点远近指标质量很低（见图 5-15）。

如何解决这个问题？很简单：如果问题来自于标度不同的特征，我们必须转换数据，使所有特征都处于可比较的相同量级。换言之，当观测值在某个特征上有低值或者高值时，对距离函数的影响应该具有可比性。

常用的方法之一是用如下公式重新标度每个特征：$x' <- (x - \min) / (\max - \min)$。结果是，每个特征都分布在 0～1 的范围内，最小值为 0，最大值为 1，其他值按照线性比例处于两者之间。举例说明，如果我们对前一个数据集应用这种方法，数据的分布如图 5-16 所示。

这不是重新标度特征的唯一方式，根据数据的不同，其他方法可能更合适。我们将在稍后看到特征的其他标度方式。特征标度是一种神秘的艺术，没有唯一的方法。经验法则是，

问问自己，你所选择的距离比较的是不是可比事物。

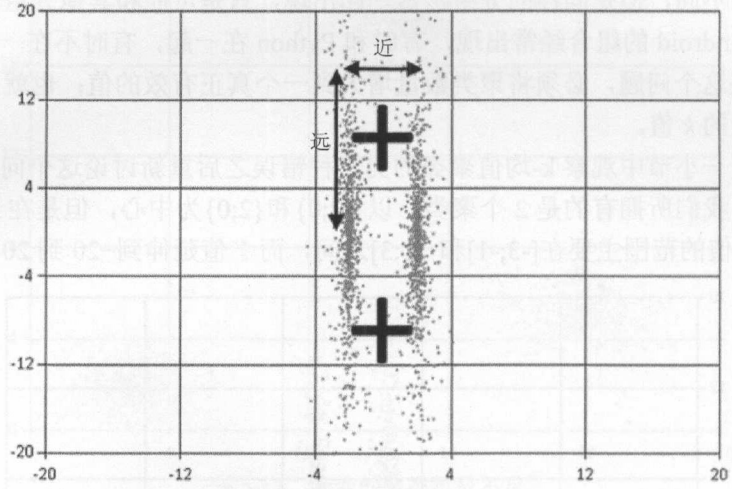


图 5-15　靠近的点

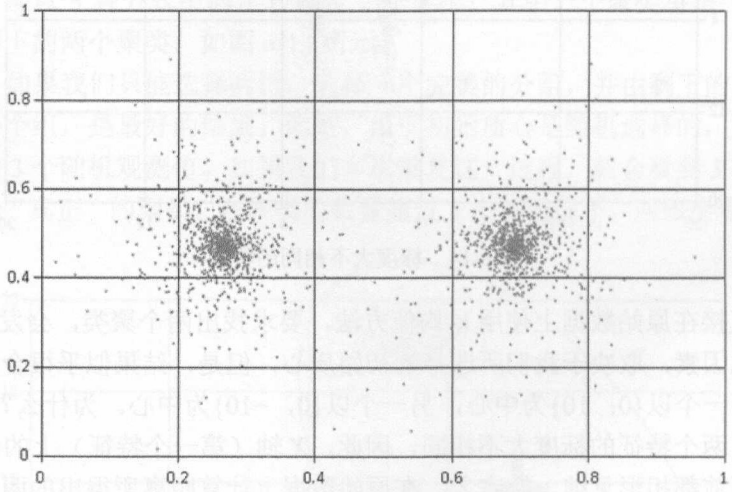


图 5-16　重新标度的特征

5.6　重新标度数据集以改进聚类

我们的初始聚类提取了似乎有意义的信息，它也有明显的问题——聚类数量不合适（每次运行算法都得到不同的结果）以及特征标度有差异。在处理确定合适聚类数量这一更困难的问题之前，我们首先从较简单的问题——数据集重新标度开始。

你可能已经发现了一个事实：最初的聚类分析发现的峰值出现在最高使用量的标签上——Javascript、Java、C#、Python 等。一般来说，虽然算法每次运行都产生不同的聚类，

但是将出现相同的一组标签。这是很有意义的，因为使用量很大的标签将在距离上产生很大的差异。这并不好，因为我们的算法本质上寻找的是高使用量标签，而不是选取较低使用量标签上更细微的差异。

这里的大问题是，每个标签上的活动量有很大差别：JavaScript 的平均使用量 20 倍于 iPhone。我们可以按照前面描述的方针重新标度各个特征，并将其映射到 0～1 的范围，但是我们将采用不同的方向。下面是对数据集的描述方法：我们认为，用户可以分类为几种兴趣配置，但是在 StackOverflow 上的活跃度水平差别很大。为了在不考虑用户在网站上活跃度的情况下直接比较其兴趣配置，我们希望消除因为活跃度引起的差异。例如，考虑这 3 个虚构的用户：

	JavaScript	C#
用户 1	100	20
用户 2	10	2
用户 3	50	50

按照原始的距离，用户 1 和用户 3 最为相似，因为他们的活跃度远高于用户 2。但是，用户 1 和用户 2 可能非常类似——比用户 1 和用户 3 更相似。虽然用户 1 的活跃度 10 倍于用户 2，但是他们将时间花费在完全相同的标签上。解决这一问题的方法之一是按照用户设置标度：对于每一行，将最大的标签标度为 100%，其他每个标签相应地除以该值。

	JavaScript	C#
User 1	100%	20%
User 2	100%	20%
User 3	100%	100%

这样，用户 1 和用户 2 突然变成完全一样的了，而用户 3 则大不相同。实际上，我们所做的是从方程中删除用户活跃度，这里隐含的模型可以这样描述：“一些 StackOverflow 用户感兴趣的标签相同。但是，其中一些人花费的时间更多，因此其活动量相应地也更高”。注意，这也意味着，我们现在忽略了用户在网站上的活跃度。这样做对吗？可能对，也可能不对——对这个问题没有明确的答案。

无论如何，我们用“坏”的聚类数量尝试一下。实际上，我们需要做的就是将观测值除以该行中的最大值。

程序清单 5-9　将观测值规格化为类似的活跃度

```
let rowNormalizer (obs:Observation) : Observation =
    let max = obs |> Seq.max
    obs |> Array.map (fun tagUse -> tagUse / max)

let observations2 =
    observations
    |> Array.filter (fun x -> Array.sum x > 0.)
    |> Array.map (Array.map float)
    |> Array.map rowNormalizer
```

```
let (clusters2, classifier2) =
    let clustering = clusterize distance centroidOf
    let k = 5
    clustering observations2 k
```

我们仍没有处理 *k* 应该是多大的问题，因此，寻找的聚类仍然可能太少：几次运行的结果仍然不稳定。但是，即便如此，我们也应该观察到几个情况。首先，聚类的大小现在平衡得多了。运行和前面一样的代码，得到的结果如下：

```
observations2
|> Seq.countBy (fun obs -> classifier2 obs)
|> Seq.iter (fun (clusterID, count) ->
    printfn "Cluster %i: %i elements" clusterID count)

>
Cluster 4: 480 elements
Cluster 5: 252 elements
Cluster 2: 219 elements
Cluster 1: 141 elements
Cluster 3: 125 elements
val it : unit = ()
```

而且，现在更清晰地出现了一些关联，如 C/C++，甚至出现了之前没有出现的配对，如 Python 和 Django，或者 Ruby 和 Ruby on Rails，这些配对中都有一个元素因为活跃度太低而无法在距离上表现出差异（见图 5-17）。

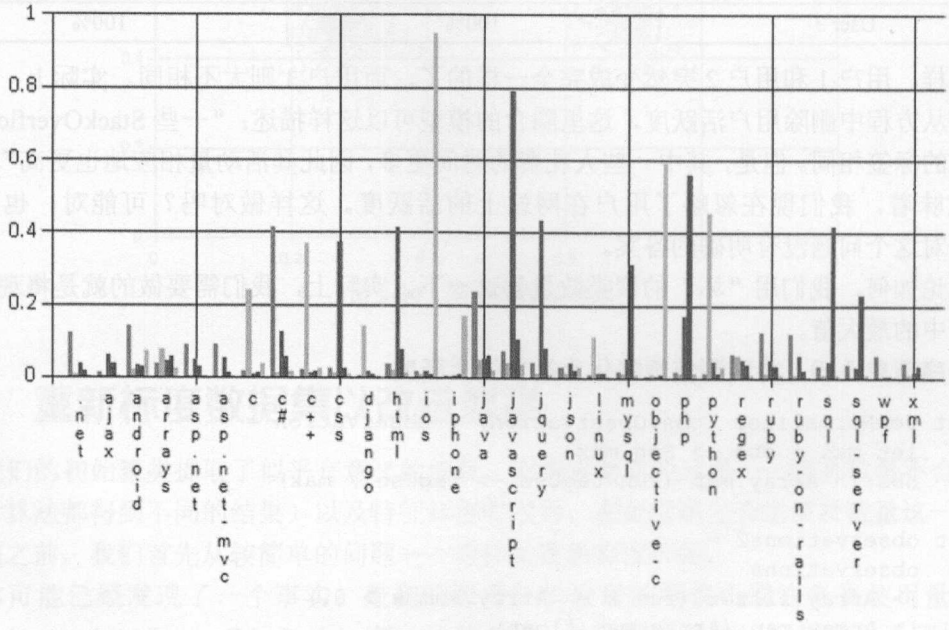


图 5-17　行规格化之后的聚类（详见源代码包）

5.7 确定需要搜索的聚类数量

我们仍然没有解决“应该搜索多少个聚类”这个关键问题。遗憾的是，正如前面我们所讨论的，这不是可以漠视的问题：搜索的聚类太少，将产生不稳定的结果。所以，我们首先设想如何解决这个问题，然后将方案应用到数据集上。

5.7.1 什么是“好”的聚类？

我将再次重申，确定聚类数量并不是简单的工作，因为关于“聚类是什么”有一定的模糊性。考虑图 5-18：你看到了几个聚类？可以认为是 3 个大的聚类，也可以认为有 7 个聚类——还可能是介乎两者之间的多种组合（为准确起见，这幅插图是用 7 个聚类生成的）。

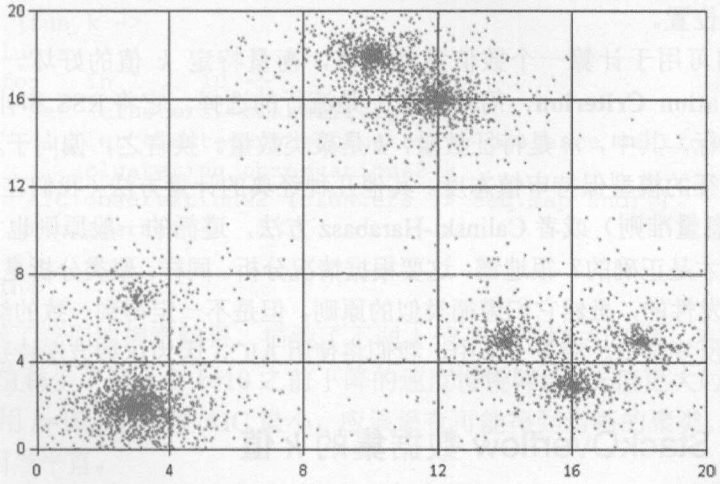


图 5-18 聚类是模糊的

我希望，上述例子至少传达了这样的意思：聚类并不完全是明确的。如果我们无法就聚类的含义达成一致或者做出解释，那么这个任务很有可能对于机械性的算法来说也是难以完成的。

那么，我们打算在聚类中找到什么呢？实际上是两个方面。聚类是相互之间有合理相似度的一组观测值，与其他聚类有可识别的差异。换言之，相对于数据集中的总体偏差，每个观测值应该接近其质心。这提示了计量质心对数据的表现能力影响是好是坏的一种方法：取得每个观测值，计量与其质心的距离，并将其加总。这种计量称作**残差平方和（Residual Sum of Squares）**，常常缩写为 **RSS**。

RSS 本质上采用的是我们在第 4 章的回归方法中已经看到的一种思路，并将其转换为聚类。在回归中，我们用一个代价函数计量模型与数据的匹配程度，计算每个数据点和对应预测值之间的平方差，并将其加总。现在，每个属于某个聚类的观测值模型化为其质心，我们通过比较每个观测值和对应的质心，计量聚类与数据的匹配程度。

尝试不同的 k 值，找出 RSS 最小的一个如何？这种方法有一个问题。假定我们有包含 N 个观测值的数据集——有一个值明显能够最小化 RSS，那就是 k=N。如果为每个观测值创建一个质心，当然可以获得完美的匹配。还要注意，如果我们有 k 个质心，增加一个质心总能降低 RSS（更准确地说，RSS 永远不会再增加）。

这有助于澄清我们所追求的目标。我们想要的是数据的一个“高效”摘要，遵循的仍然是“奥卡姆剃刀”原则。我们希望聚类能够很好地解释数据，并且尽可能简单。如果只考虑 RSS，就遗漏了第二个要点：如果对复杂度没有任何惩罚，聚类将总是过度拟合，简单地重现数据集而不是任何形式的摘要。

那么，我们应该做什么？一般的回答是，如果有聚类质量的计量指标，应该包含 RSS（模型对数据的拟合质量）以及复杂性的补偿项。同样，这是一个困难的问题，没有任何万能的高招。有几种流行的方法。首先，想象你已经计算了一个 k 值的 RSS。现在，计算 k+1 值的 RSS：如果你只打算改善 RSS，增加聚类数量并不经济。这种方法称作“弯头”——寻找 RSS 相对改善降级的位置。

这个大方向可用于计算一个数值型计量值，衡量特定 k 值的好坏。**赤池信息量准则**（**Akaike Information Criterion**，简称 **AIC**）是流行的选择。它将 RSS 和补偿项 $2\times m\times k$ 组合为一个计量指标，其中，m 是特征数量，k 是聚类数量。换言之，偏向于聚类较少的模型，对于具有许多特征的模型保持审慎态度。其他几种经典的计量方法（我们在此不予实施），如 BIC（贝叶斯信息量准则）或者 Calinski-Harabasz 方法，遵循的一般原则也大致相同。

哪一种方法才是正确的？很遗憾，这要根据情况分析。同样，聚类分析是有些模糊的任务，所有方法都是启发性的，虽然它们遵循类似的原则，但是不一定得到一致的结果。确定某种方法有效通常需要反复尝试。在下一节中，我们将使用 AIC，因为这种方法特别容易实现。

5.7.2　确定 StackOverflow 数据集的 k 值

让我们来看看前述方法在手上的数据集中的应用。我们用于调整 k 值的第一个启迪是工业界常用的一个“经验法则”，该法则建议使用如下 k 值：

```
let ruleOfThumb (n:int) = sqrt (float n / 2.)
let k_ruleOfThumb = ruleOfThumb (observations2.Length)
```

很明显，这种方法过于简单，无法很好地用于各种场合，它所考虑的唯一事实是数据集的大小！另一方面，它具有简洁的好处。我推荐使用这种方法大致了解合理值的范围。在本例中，建议值为 25，我们将其作为 k 值的上界。下面，我们尝试 AIC 方法。

程序清单 5-10　赤池信息量准则（AIC）

```
let squareError (obs1:Observation) (obs2:Observation) =
    (obs1,obs2)
    ||> Seq.zip
    |> Seq.sumBy (fun (x1,x2) -> pown (x1-x2) 2)
```

```
let RSS (dataset:Observation[]) centroids =
    dataset
    |> Seq.sumBy (fun obs ->
        centroids
        |> Seq.map (squareError obs)
        |> Seq.min)

let AIC (dataset:Observation[]) centroids =
    let k = centroids |> Seq.length
    let m = dataset.[0] |> Seq.length
    RSS dataset centroids + float (2 * m * k)
```

为了找出 k 的合理值，我们将尝试 1～25 之间的所有可能性。因为聚类本身在 k 值较低时不稳定，我们将对每个值运行算法数次，得出对应 AIC 的平均值，以便消除由于不走运的初始值选择等原因造成的潜在偶然现象：

```
[1..25]
|> Seq.map (fun k ->
    let value =
        [ for _ in 1 .. 10 ->
            let (clusters, classifier) =
                let clustering = clusterize distance centroidOf
                clustering observations2 k
            AIC observations2 (clusters |> Seq.map snd) ]
        |> List.average
    k, value)
|> Chart.Line
```

图 5-19 显示了我们的搜索结果，描绘了不同 k 值下的平均 AIC 值。在 k 从 1 增加到 5 的过程中，AIC 值稳步下降，在 k=10 之前下降的速度慢慢降低，k 值更大时则开始回升。这说明我们应该使用 k=10，该值使 AIC 最小，应该最有可能得到清晰的聚类。但是要注意，该曲线在 10 周围相当平直。

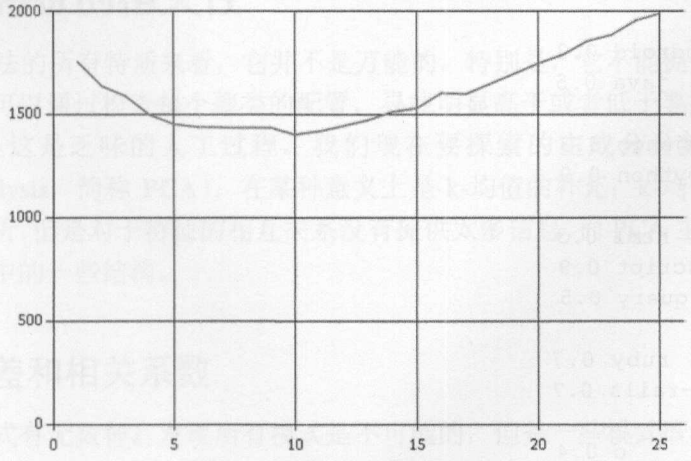


图 5-19 搜索 AIC 最小的 k 值

5.7.3　最终的聚类

我们已经根据 AIC 为 k 选择了值 10，现在来看看发生了什么。现在，可以运行聚类算法多次以避免偶然性，选择特定 k 值下 RSS 最小的模型。

程序清单 5-11　最终的聚类

```
let (bestClusters, bestClassifier) =
    let clustering = clusterize distance centroidOf
    let k = 10
    seq {
        for _ in 1 .. 20 ->
            clustering observations2 k
    }
    |> Seq.minBy (fun (cs,f) ->
        RSS observations2 (cs |> Seq.map snd))
```

聚类中有什么？检查一下，然后打印出每个聚类中最高使用量的标签。注意，一般来说，只选择高值并不是好主意：我们真正寻找的是明显不同于其余观测值的数值。通常，选择与平均值的最大绝对差值可能更好，但是在我们的例子中，平均值很小，因此可以简单地寻找异常的高值。

```
bestClusters
|> Seq.iter (fun (id,profile) ->
    printfn "CLUSTER %i" id
    profile
    |> Array.iteri (fun i value ->
        if value > 0.2 then printfn "%16s %.1f" headers.[i] value))

>
CLUSTER 5
         android 0.3
            java 0.5
CLUSTER 7
          django 0.3
          python 0.9
CLUSTER 4
            html 0.3
      javascript 0.9
          jquery 0.5
CLUSTER 6
            ruby 0.7
   ruby-on-rails 0.7
CLUSTER 10
               c 0.4
             c++ 0.8
```

```
CLUSTER 2
      javascript 0.3
           mysql 0.2
             php 1.0
CLUSTER 8
            .net 0.3
              c# 0.9
CLUSTER 9
           mysql 0.3
             sql 0.8
      sql-server 0.6
CLUSTER 1
             ios 1.0
    objective-c 0.6
CLUSTER 3
             css 0.9
            html 0.7
      javascript 0.5
          jquery 0.3
```

结果看上去不错。我想你应该能很轻松地在心中描绘每个聚类的配置，这明显对应于不同的开发堆栈。这是了不起的成就：没有花费太大的精力，就自动地从原始数据中提取了有意义的信息。没有对标签含义的任何理解，只凭对数据集和历史使用量模式的观察，k-均值方法就领悟到 Ruby 和 Ruby on Rails 的相互关联，或者说我们称为 DBA 的人们主要专注的 SQL 相关技术。

使用 k-均值方法还有可能从数据集中榨出更多信息，但是，如果继续朝着这个方向前进，获得新的有趣结果将越来越难。我们对 k-均值的介绍到此为止，转而介绍一种完全不同的技术——主成分分析。我们将把它应用到同一个问题上，以便比较两种方法。

5.8 发现特征的相关性

从 k-均值方法的所有特质来看，它并不是万能的。特别是，它不能提供特征相互作用的全面图景。我们可以通过检查每个聚类的配置，寻找明显高于或者低于典型值的特征，找出某些关系，但是这是乏味的人工过程。我们现在要探索的**主成分分析方法**（**Principal Component Analysis**，简称 **PCA**），在某种意义上是 k-均值的补充：k-均值清晰地提供了对单独观测值的剖析，但是对于特征的相互关系没有提供太多信息，而 PCA 主要是发现特征（而非单独观测值）中的一些结构。

5.8.1 协方差和相关系数

数据中的模式有无数种。发现所有模式是不可能的，但是一些模式既实用又容易发现。下面是一个例子：“每当数量 A 增加，质量 B 也增加”。这就是**协方差**和**相关系数**的含义。

两个计量值 X 和 Y 之间的协方差定义为：

```
Cov(X,Y) = average [(X-average(X)) * (Y-average(Y))]
```

当你开始分解上述公式时，就会看到一条合理的路径。如果两个值 *X* 和 *Y* 同时上升和下降，则一个值高于其平均值时，另一个值也是如此，其乘积将为正数。同样，当一个值低于其平均值时，另一个值也是如此——与平均值差值（现在皆为负数）的乘积仍然为正数。根据相同的原因，可以预期两个变化方向相反的特征的协方差为负数。

附注：协方差（或者相关系数）为 0 并不意味着两个特征之间没有关系，理解这一点很重要。这只是说明，它们的关系不是简单的同时上升或下降。举个例子，描述圆形的数据点的相关系数为 0，但是它们的关系完全可以确定。

协方差是很出色的工具，但是有一个问题：它依赖于特征的标度，自身没有明显的标度，因此很难诠释。什么是“高”的协方差？这取决于数据。相关系数是协方差的改良形式，解决了这个问题。特征 X 和 Y 的相关系数的正式定义如下：

```
Corr(X,Y) = Cov(X,Y) / (stdDev(X) * stdDev(Y))
```

式中的 stdDev 是**标准差**（**standard deviation**）的缩写。基于你对统计学的熟悉程度，这一公式看起来似乎很可怕。无须担心——我们将稍做分解。协方差的问题在于 X 和 Y 之间的标度差异。为了使不同的特征可以比较，我们需要的是将(X−average(X))和(Y−average(Y))两项归约为类似的标度。将(X−average(X))除以其平均值——X 与其平均值之间的平均距离（真拗口！）——就可以实现这个目标，这就是标准差的含义（常标记为 Σ）。

提示：标准差（及其平方——方差）往往被描述为特征分散程度的计量。我个人觉得如下的描述很有帮助：**标准差 Σ 计量与平均值的平均距离**。如果从样本中随机取一个观测值，标准差是它与样本均值的距离。

可以扩展方差部分，将公式重组为下面的等价版本，这有助于我们更好地理解相关系数的作用：

```
Corr(X,Y) = average [(X-avg(X))/stdDev(X) * (Y-avg(Y))/stdDev(Y)]
```

(X−average(X))/stdDev(X)项取协方差的平均值并用 Σ 重新标度，平均差值现在为 1。X 和 Y 项现在处于可比的标度，结果是（在此不做证明），相关系数的属性令人愉快——范围为−1 和 1 之间。1 代表 X 和 Y 以完全线性关系同向变化，−1.0 表示方向相反的完全线性关系，0 表示没有发现上述两种关系。相关系数越靠近 1 或者−1，模式越明显——也就是说，特征几乎是同时变化的，但是有某种程度的不确定性。

顺便说一句，将计量值 X 转换为(X−average(X))/stdDev(X)是重新标度某个特征的经典方法之一，被称作 **z-得分**。这种方法比我们之前讨论的方法（简单地将数值从[min;max]转换为[0;1]）更复杂，但是有一些优点。转换后的特征以 0 为中心，所以正负值可以直接解释为高

或者低。如果特征的分布合理，也就是围绕均值呈钟形分布，这一结果也就更加“合理”。从定义上看，最小值和最大值是极端的异常值，而标准差通常是用于标度观测值的更好指标。

5.8.2 StackOverflow 标签之间的相关性

让我们继续前进，看看上述概念在数据集中的应用。为了简单起见，我们不在当前脚本中添加代码，而是创建一个新脚本 PCA.fsx 和一个模块 PCA.fs，主成分分析代码将从该模块中提取。在本节中，我们还将使用一些线性代数和统计学知识，因此通过 NuGet 在项目中添加对 Math.NET Numerics 和 Math.NET Numerics for F#的引用，以避免重复劳动。脚本的开头和前面一样，加载对 fsharp.Charting 的引用，并读取文件中的表头和观测值。

事实上，Math.Net 有一个内建的相关矩阵。Correlation 函数需要的数据采用特征而非观测值的形式——也就是说，不是 1600 个观测点（每个包含 30 个值），而是需要将其转置为对应 30 个特征的 30 行。

程序清单 5-12 计算相关矩阵

```
#r @"..\packages\MathNet.Numerics.3.5.0\lib\net40\MathNet.Numerics.dll"
#r @"..\packages\MathNet.Numerics.FSharp.3.5.0\lib\net40\MathNet.Numerics.FSharp.dll"

open MathNet
open MathNet.Numerics.LinearAlgebra
open MathNet.Numerics.Statistics

let correlations =
    observations
    |> Matrix.Build.DenseOfColumnArrays
    |> Matrix.toRowArrays
    |> Correlation.PearsonMatrix

>

val correlations : Matrix<float> =
 DenseMatrix 30x30-Double
          1     0.00198997  -0.00385582 0.11219 0.425774 .. 0.172283 0.0969075
 0.00198997              1  -0.0101171 0.259896 0.16159 .. -0.00783435 0.025791
-0.00385582   -0.0101171   1 0.0164469 -0.00862693 .. -0.00465378 0.0775214
// snipped for brevity
```

正如预期，我们得到了一个 30×30 矩阵，每个元素对应于两个特征之间的相关系数。值得欣慰的是，对角线元素都为 1（特征与自身应该是完全相关的），矩阵是对称的（A 和 B 以及 B 和 A 的相关系数应该相等）。这并不特别有用，因为数据太多，我们要取得每个标签对，从矩阵中得到其相关系数，提取 20 个绝对值最大的，因为大的正负值表示同样强的相关：

```
let feats = headers.Length
let correlated =
    [
        for col in 0 .. (feats - 1) do
            for row in (col + 1) .. (feats - 1) ->
                correlations.[col,row], headers.[col], headers.[row]
    ]
    |> Seq.sortBy (fun (corr, f1, f2) -> - abs corr)
    |> Seq.take 20
    |> Seq.iter (fun (corr, f1, f2) ->
        printfn "%s %s : %.2f" f1 f2 corr)

>
ios objective-c : 0.97
ios iphone : 0.94
mysql sql : 0.93
iphone objective-c : 0.89
sql sql-server : 0.88
css html : 0.84
.net c# : 0.83
javascript jquery : 0.82
ajax javascript : 0.78
mysql sql-server : 0.76
html javascript : 0.73
html jquery : 0.71
ajax jquery : 0.70
ajax html : 0.66
ajax json : 0.60
javascript json : 0.56
asp.net c# : 0.53
c c++ : 0.50
ruby ruby-on-rails : 0.50
mysql php : 0.48
```

提示：在本书编著的时候，F#中没有内建的降序排序功能。作为替代，你可以按照相反数排序；例如 Seq.sortBy (fun (corr, f1, f2) -> - abs corr)。

结果非常合理。我们立刻看出一些关系紧密的特征组：iOS、objective-c 和 iPhone；SQL、MySQL 和 SQL-server 等。花费少量精力，相关矩阵就从数据集中提取了许多信息。

5.9　用主成分分析确定更好的特征

我们将相关矩阵作为特征间关系的实用总结来看待。另一种诠释方式是将其视为数据集

包含许多冗余信息的信号。举个例子，iOS、objective-C 和 iPhone 的相关系数很靠近 100%。这意味着我们几乎可以将这 3 个特征合而为一——比如“iOS 开发人员”。一旦知道了 3 个特征中的一个很高（或者很低），就可以知道另两个特征的情况。

与此同时，有些相关性较难处理。例如，如果深入研究数据集，就可以看出 MySQL 和 SQL、SQL 和 SQL-Server、MySQL 和 PHP 之间都有强相关。这是否意味着通过关系的传递，PHP 和 SQL-Server 也相关？这值得怀疑——从我的个人经验看，这两个关系应该视为不同的关系，一边可能是 DBA，另一边则可能是 LAMP 开发人员。他们碰巧在 MySQL 上有共同的兴趣，但是并不是同一群人。

在本节中，我们将阐述主成分分析（Principd Component Analysis，简称为 PCA），这是一种数据挖掘技术，观察协方差或者相关矩阵的结构，并利用它们将数据重组为一组新的特征，将现有特征组合为更有效地代表数据、避免冗余的混合体。

5.9.1 用代数方法重新组合特征

全面解释 PCA 的工作原理需要深入代数理论，这些知识最终没有太多的用处。我们将通过例子说明关键的要点，在具有战略性的时刻有意地做些强调，让感兴趣的读者自行深入了解。

我们所追求的，是一种重新组织数据集的手段。例如，考虑一个只有两个特征（而不是我们具有 30 个特征）的数据集。表示特征变换的方式之一是使用一个 2×2 矩阵。如果将一个观测值（两个元素组成的向量）乘以该矩阵，就可以得到一个新向量，该向量仍然有两个特征：

```
let observation = vector [ 2.0; 5.0 ]
let transform =
    matrix [ [ 1.0; 3.0 ]
             [ 2.0; 1.0 ] ]
transform * observation |> printfn "%A"

>
seq [17.0; 9.0]
```

变换矩阵中的每一行可以视为权重。例如，第一行表明第一个转换后的特征中包含 1 份第一个原特征，以及 3 份第二个特征。如果变换使用单位矩阵（对角线为 1，其余元素为 0），返回的就是原来的特征。

我们所寻求的就是这样一个矩阵。显然，不能简单地选择任意矩阵，随机矩阵只是重新布置特征，但是没有理由产生任何特别有用的组合。那么，如何才能实现这样的变换呢？在这种特殊的场合下，我们希望转换重新安排特征，限制相关矩阵中发现的冗余信息，还希望有一种方法能够区别数据集中的重要趋势和不重要的差异。

这就是需要强调的部分。事实上，一种线性代数技术能够完成我们所需的工作：将矩阵分解为特征向量和特征值。概略来讲，$N \times X$ 矩阵 M 可以唯一分解为 N 对特征值和特征向量

（一个浮点数和一个 N 元素向量），每一对都满足如下等式：$\boldsymbol{M}$×特征向量= 特征值×特征向量。换言之，特征向量是将 $\boldsymbol{M}$ 延伸到与其特征值对应的倍数的一个方向。

如果将这一技术应用到协方差矩阵，将得到什么？首先，每个特征向量代表组合现有特征形成的一个新特征。记住，协方差矩阵计量每个特征与平均值的差值，具有最大特征值的特征向量代表延伸程度最大的特征。因此，在协方差矩阵的情况下，这意味着我们观察到的是数据中心的最大协变方向。换个说法，具有最大特征值的特征变量是组合现有特征，能够解释数据集中观察到的大部分差异的一个特征。在消除第一个特征的影响之后，第二大的特征值隔离了下一个信息量最大的组合，依此类推。

5.9.2　PCA 工作方式预览

到目前为止，一切都还相当抽象。在实施 PCA 并应用到整个数据集之前，我们先用一个小例子说明工作目标。我们将使用很快就要编写的代码，忽略细节，简单地说明预期的结果类型，澄清 PCA 的工作方式。

前面进行的相关性分析说明，两对特征之间有强相关性：iOS 和 Objective-C，SQL 和 SQL Server。让我们忽略其他所有特征，只在这 4 个特征上进行主成分分析。

分析提取了 4 个特征值，量值分别为 98.73、38.59、5.69 和 0.61，总计 143.61。这些值代表了每个新特征解释的原始信息量。例如，第一个（最大的）成分足以捕捉数据变化的 98.73 / 143.61≈69%，第二个成分可以捕捉 27%。换言之，只使用两个主要特征——两个主成分——就可以重现原始数据集中 95%以上的信息。最后两个成分只代表不到 5%的信息，可以忽略，因为它们几乎不能带来任何附加信息。

这些新特征是什么样子呢？只需观察特征向量便可得知，表 5-1 以表格形式显示了特征向量。每列对应一个成分，按照重要性降序排列。每一行对应一个原始特征，每个值代表了原始特征在成分中的“权重”。

表 5-1　　主成分

特征	成分 1	成分 2	成分 3	成分 4
iOS	0.00	−0.70	0.00	−0.71
Objective-C	0.00	−0.71	0.00	0.70
SQL	0.80	0.00	−0.59	0.00
SQL Server	0.59	0.00	0.80	0.00

在这个例子中，我们看到成分 1 组合了 SQL 和 SQL Server，成分 2 组合了 Objective-C 和 iOS。这很好：PCA 本质上将数据集从 4 个特征简化为 2 个特征——“iOS /Objective-C”和“SQL / SQL Server”，几乎没有丢失任何信息。

我们可以表示原始特征在主成分上的映射，可视化上述信息，如图 5-20 所示。这在特征数量增加的情况下特别方便，可以发现数字表格中难以发现的特征间的关系。

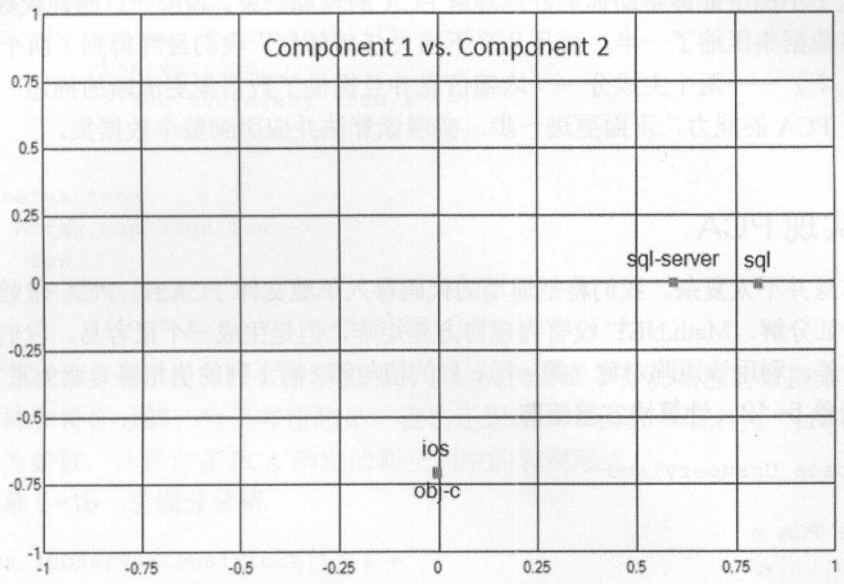


图 5-20 投影到主成分上的原始特征

另一种图表也常常用到，在原始观测值从原始特征转换为主成分之后对其进行可视化。举个例子，图 5-21 显示，StackOverflow 用户通常不会同时关心两个主题（只有一个例外），积极参与 SQL 讨论的人——也就是图表右侧的观测值——不关心 iPhone 开发，反之亦然。

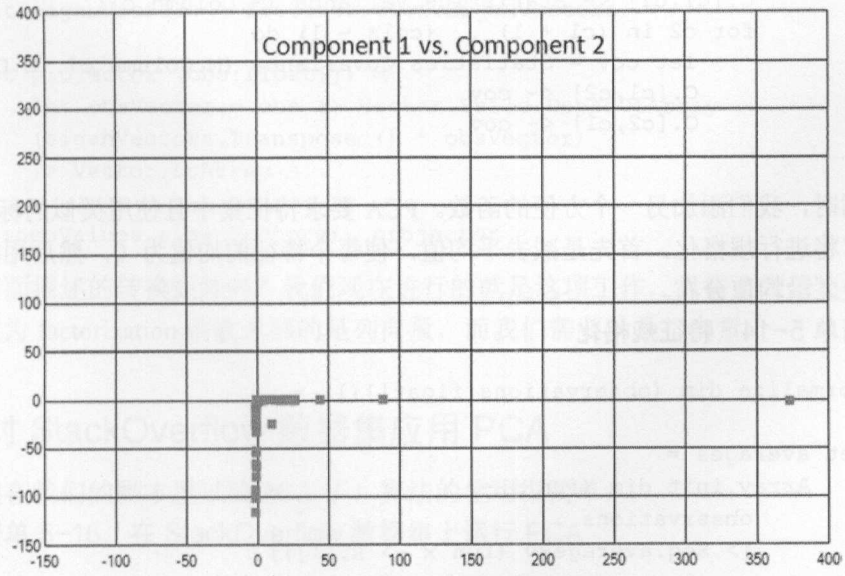


图 5-21 投影到主成分上的原始观测值

希望这个小例子能够帮助你更好地理解 PCA 的预期结果。利用一点线性代数的知识，我们本质上将数据集压缩了一半，而且几乎不损失任何信息。我们最终得到了两个特征（而不是原来的 4 个）——两个主成分——浓缩信息并且提供了数据集更清晰的描述。现在，我们已经看到了 PCA 的能力，下面更进一步，实现该算法并应用到整个数据集。

5.9.3　实现 PCA

实现本身并不太复杂。我们将把通用的代码移入单独文件 PCA.fs。PCA 依赖数据集协方差矩阵的特征分解。Math.NET 没有内建协方差矩阵，但是生成一个很容易。我们计算每个特征列的协方差，利用该矩阵对称（第 r 行 c 列的值与第 c 行 r 列的值相等）避免重复的工作。

程序清单 5-13　计算协方差矩阵

```
namespace Unsupervised

module PCA =

    open MathNet
    open MathNet.Numerics.LinearAlgebra
    open MathNet.Numerics.Statistics

    let covarianceMatrix (M:Matrix<float>) =
        let cols = M.ColumnCount
        let C = DenseMatrix.create cols cols Matrix.Zero
        for c1 in 0 .. (cols - 1) do
            C.[c1,c1] <- Statistics.Variance (M.Column c1)
            for c2 in (c1 + 1) .. (cols - 1) do
                let cov = Statistics.Covariance (M.Column c1, M.Column c2)
                C.[c1,c2] <- cov
                C.[c2,c1] <- cov
    C
```

与此同时，我们添加另一个方便的函数。PCA 要求特征集中且使用类似的标度。为此，数据集通常将进行规格化，首先是减去平均值，使每个特征的均值为 0，然后用标准差缩小特征值，使之沿均值分布。

程序清单 5-14　特征规格化

```
let normalize dim (observations:float[][]) =

    let averages =
        Array.init dim (fun i ->
            observations
            |> Seq.averageBy (fun x -> x.[i]))

    let stdDevs =
```

```
        Array.init dim (fun i ->
            let avg = averages.[i]
            observations
            |> Seq.averageBy (fun x ->
                pown (float x.[i] - avg) 2 |> sqrt))

    observations
    |> Array.map (fun row ->
        row
        |> Array.mapi (fun i x ->
            (float x - averages.[i]) / stdDevs.[i]))
```

主成分分析本身几乎没有什么需要实现的了。从包含许多行观测值的数据集中，我们计算协方差矩阵和特征分解。为了方便起见，我们还返回一个函数（取名为“projector”），以单独观测点为参数，计算它在 PCA 确定的新空间中的表现形式。

程序清单 5-15　主成分分析

```
let pca (observations:float[][]) =

    let factorization =
        observations
        |> Matrix.Build.DenseOfRowArrays
        |> covarianceMatrix
        |> Matrix.eigen

    let eigenValues = factorization.EigenValues
    let eigenVectors = factorization.EigenVectors

    let projector (obs:float[]) =
        let obsVector = obs |> Vector.Build.DenseOfArray
        (eigenVectors.Transpose () * obsVector)
        |> Vector.toArray

    (eigenValues,eigenVectors), projector
```

记得前面描述的转换矩阵吗？我们现在进行的就是这项工作，只需要转置特征向量矩阵就行了，因为 factorization 函数返回的是列向量，而我们需要的是行向量。

5.9.4　对 StackOverflow 数据集应用 PCA

是时候在我们的脚本里试验 PCA 了！算法的使用很简单：

程序清单 5-16　在 StackOverflow 数据集上运行 PCA

```
#load "PCA.fs"
open Unsupervised.PCA
```

```
let normalized = normalize (headers.Length) observations

let (eValues,eVectors), projector = pca normalized
```

观察输出，对特征值的检查很常用，可以提供每个主成分（新特征）包含的信息量。为此，我们将计算提取的 30 个特征值的总量，然后输出每个特征的占比，以及仅使用信息量最大的特征时覆盖的总量。

程序清单 5-17　特征权重分析

```
let total = eValues |> Seq.sumBy (fun x -> x.Magnitude)
eValues
|> Vector.toList
|> List.rev
|> List.scan (fun (percent,cumul) value ->
    let percent = 100. * value.Magnitude / total
    let cumul = cumul + percent
    (percent,cumul)) (0.,0.)
|> List.tail
|> List.iteri (fun i (p,c) -> printfn "Feat %2i: %.2f%% (%.2f%%)" i p c)

>
Feat 0: 19.14% (19.14%)
Feat 1: 9.21% (28.35%)
Feat 2: 8.62% (36.97%)
// snipped for brevity
Feat 28: 0.07% (99.95%)
Feat 29: 0.05% (100.00%)
```

这与相关系数保持一致：前 5 个特征可以解释数据集的 50%以上，前 10 个的覆盖面接近 80%。相比之下，最后 10 个特征每个仅覆盖不到 1%的信息。

警告：按照特征值降序查看特征是习惯的做法，但是 Math.NET 返回的特征值和特征向量采用相反的顺序。因此，最重要的主成分将出现在特征向量矩阵的最后一列。一定要小心！

5.9.5　分析提取的特征

我们可以观察一下这些特征吗？每个特征值直接将旧特征映射到一个新特征（主成分）。我们需要的就是从前面的 PCA 得到的分析结果中捕捉特征向量，将每个特征向量的值映射到对应的标签名称。程序清单 5-18 用于绘制原始特征与任何一对成分的对比图表。注意，我们在重建成分 x 时，读取的是特征向量的第 30-x 列，因为分析按照重要性递增顺序返回。

程序清单 5-18　绘制原始特征与提取成分的对比图表

```
let principalComponent comp1 comp2 =
    let title = sprintf "Component %i vs %i" comp1 comp2
```

```
        let features = headers.Length
        let coords = Seq.zip (eVectors.Column(features-comp1)) (eVectors.Column
features-comp2))
        Chart.Point (coords, Title = title, Labels = headers, MarkerSize = 7)
        |> Chart.WithXAxis(Min = -1.0, Max = 1.0,
            MajorGrid = ChartTypes.Grid(Interval = 0.25),
            LabelStyle = ChartTypes.LabelStyle(Interval = 0.25),
            MajorTickMark = ChartTypes.TickMark(Enabled = false))
        |> Chart.WithYAxis(Min = -1.0, Max = 1.0,
            MajorGrid = ChartTypes.Grid(Interval = 0.25),
            LabelStyle = ChartTypes.LabelStyle(Interval = 0.25),
            MajorTickMark = ChartTypes.TickMark(Enabled = false))
```

然后，我们可以在 F# interactive 中调用 principalComponent 1 2;;，可视化两个最重要的主成分。图 5-22 显示了 PCA 识别的一些最大特征——成分 1 和成分 2 以及成分 3 和成分 4 的对比。不出意外，在给定的相关矩阵下，主要特征会非常清晰地映射到 SQL、MySQL 和 SQL-Server，与任何其他特征则毫无关系，我们将这个主成分称为“DBAs”。第二个新特征较难解读，似乎 WPF 与许多其他主题（特别是 Python/Django 和 Ruby）发展方向相反，与 C#和.NET 相关的技术（如 ASP.NET 和 ASP.NET MVC）也是如此，但是程度较低。特征 3 也不是很明确，但是 Web 和非 Web 主题方向相反，Ruby on Rails、Django、HTML 和 Javascript 在同一侧。特征 4 明显地表现出 Python 和 Ruby 的对抗。

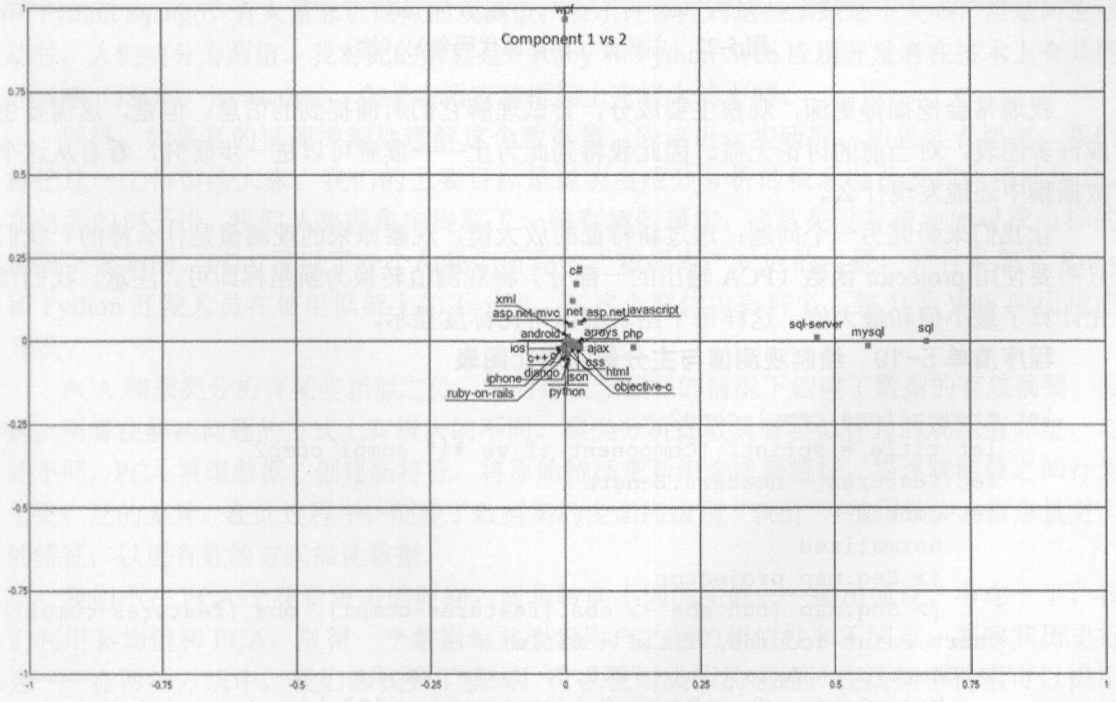


图 5-22 主成分（详见源代码包）

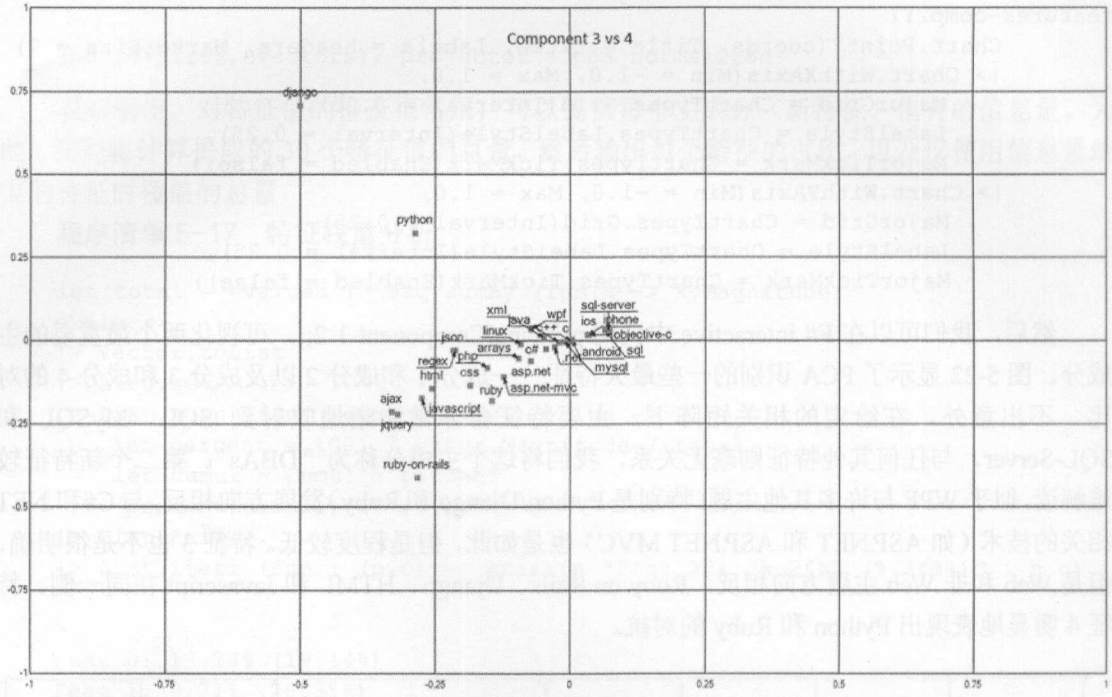


图 5-22　主成分（详见源代码包）（续）

我通常会挖掘得更深，观察主要成分，尝试理解它们所捕捉到的信息。但是，这需要生成许多图表，对当前的讨论无益。因此我将到此为止——读者可以进一步研究，看看从这个数据集中还能发现什么。

让我们来研究另一个问题：透过新特征的放大镜，观察原来的观测值是什么样的？我们只需要使用 projector 函数（PCA 输出的一部分）将观测值转换为新坐标即可。注意，我们预先计算了最小值和最大值，这样每个图表都以可比标度显示。

程序清单 5-19　绘制观测值与主分量的对比图表

```
let projections comp1 comp2 =
    let title = sprintf "Component %i vs %i" comp1 comp2
    let features = headers.Length
    let coords =
        normalized
        |> Seq.map projector
        |> Seq.map (fun obs -> obs.[features-comp1], obs.[features-comp2])
    Chart.Point (coords, Title = title)
    |> Chart.WithXAxis(Min = -200.0, Max = 500.0,
        MajorGrid = ChartTypes.Grid(Interval = 100.),
        LabelStyle = ChartTypes.LabelStyle(Interval = 100.),
```

```
        MajorTickMark = ChartTypes.TickMark(Enabled = false))
    |> Chart.WithYAxis(Min = -200.0, Max = 500.0,
        MajorGrid = ChartTypes.Grid(Interval = 100.),
        LabelStyle = ChartTypes.LabelStyle(Interval = 100.),
        MajorTickMark = ChartTypes.TickMark(Enabled = false))
```

图 5-23 显示了和前面一样的两对成分的结果。这些图不是最令人兴奋的，但是仍然显示了一些有趣的模式。第一个图表在 X 轴上绘制“数据库”成分，在 Y 轴上绘制 WPF/.NET 成分。首先，大部分观测值都明显地落在各自的坐标轴上，在原点附近聚集。这说明关心某个主题的人们通常不关心其他主题（在 SQL 上得到高分的人们在其他坐标轴上得分不高，反之亦然），大部分人在某个主题上也并不活跃。而且，观测值的分布不均匀：少数观测值得分很高，大量观测值接近于 0，少数介于两者之间。考虑到我们建立数据集时收集了每个标签中最活跃的人，这个结果并不意外：我们应该会看到某些用户的得分远高于平均水平。

提示：图 5-23 展示了一些大的“异常值”——远离平均值的观测值。异常值是数据集中需要注意的潜在问题。几个大的异常值可能会造成模型失真，这种模型对普通观测值的代表性较差。

第二个图很有趣，看上去一个坐标轴表示的是 Web 开发，另一个坐标轴则是 Ruby/Rails 和 Python/Django。有大量靠近原点的观测值，表示许多人对这些主题都不关心，但是向左移动时，人们被分为两组。我对此的解读是，Ruby 和 Python Web 应用开发者在技术上有共同的兴趣（JSON，javascript），但是在语言和框架上有极大的不同。

同样，如果真的试图深刻地理解这个数据集，应该进一步研究，如果读者愿意，我们将把这一工作留给大家。我们的主要目标是说明主成分分析的概念以及发挥作用的场合。在前面的例子中，我们从数据集中提取了一些有趣的事实，这是在没有指定所寻求目标的情况下做到的。PCA 发现了一个与我们所称的“数据库”对应的主题，并且发现了 Ruby 和 Python 开发人员在使用框架上的不一致，在这个群体中有许多人致力于 Web 应用程序开发。

PCA 和聚类分析有某些相似之处，它们都在无监督的情况下创建了数据的有效摘要。但是，两者在解决问题的方式上有极大的不同。聚类分析提取具有类似行为的观测值分组，与此不同，PCA 重组数据，创建新特征，将原始特征重新组合成新特征，描述观测值之间行为上更广泛的差异。在此过程中，创建了数据集的全面高级别“映射”，使用较少、信息量更大的特征，以更有效的方式描述数据。

我们不在 PCA 上花费更多的时间，而是转向不同的主题——提出建议。考虑一下，我们利用 k-均值和 PCA，取得一个数据集并搜索用户之间的相似性和不同点，观察其历史行为——在两种方法中，我们都找到了模式。可以使用类似的思路，在数据中搜索可以提出建议的模式——例如，向用户推荐他们可能感兴趣的标签。

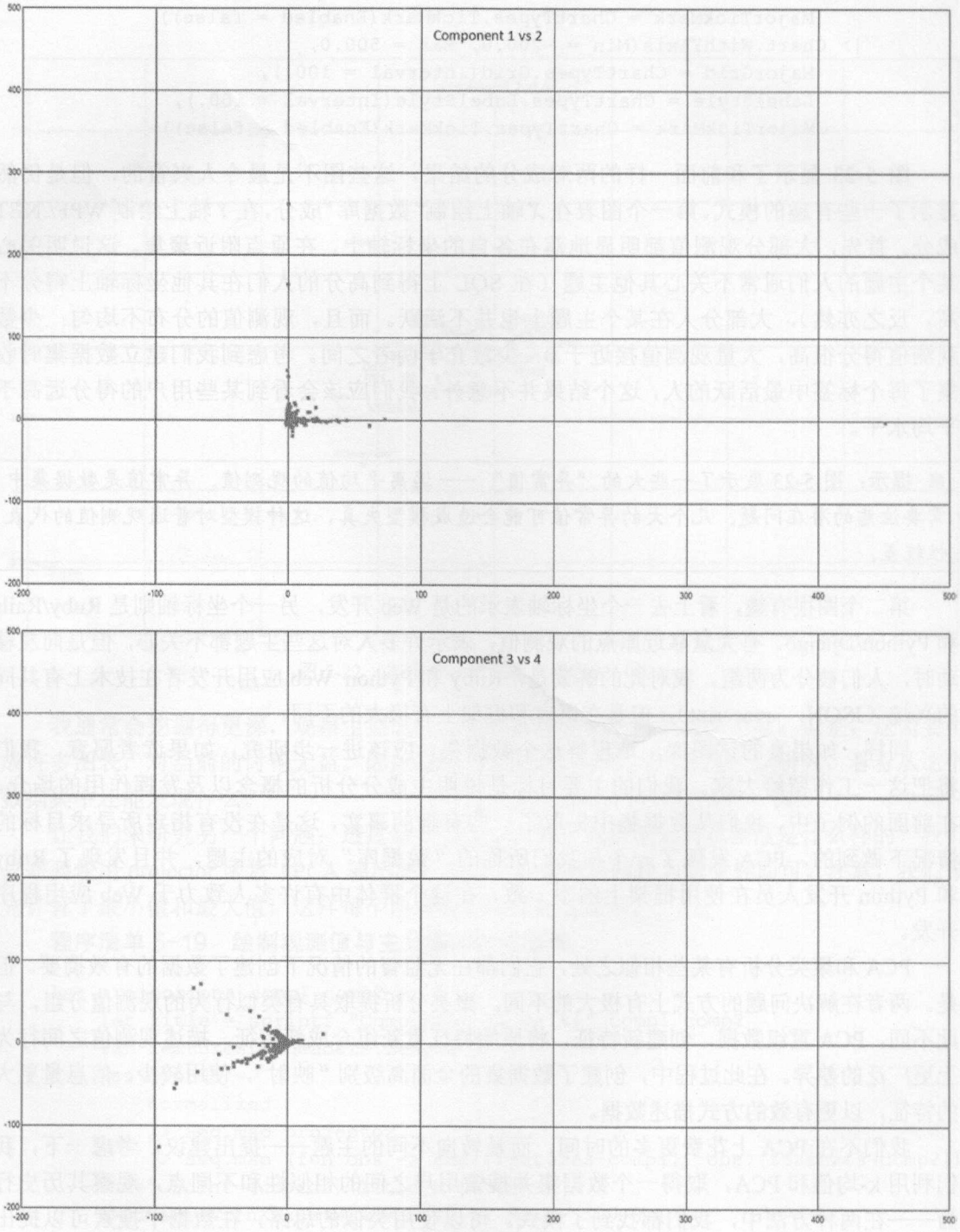


图 5-23　原始观测值与主成分的对比

5.10 提出建议

回顾我们用聚类和主成分分析所进行的工作，在两种方法中，我们观察了许多个人在一段时间内的行为，搜索他们的相似性。两种方法都取得了效果。我们都有自己的个性，但是完全与众不同的人毕竟很少。只要观察的人足够多，就会找到和你很相似的人。

这就是**协同过滤**的基本思路。简言之，这一思路就是：如果你和我都表达了对一组物品的相同偏好，就可能在其他事物上也有类似的品味。例如，如果我们恰巧喜欢或不喜欢同样的 10 部电影，而且我知道你喜欢我尚未看过的一部电影，那么认为我也喜欢这部电影是有道理的，你建议我去看这部电影合情合理。

当然，这也有可能只是个巧合。可能我们都喜欢的这 10 部电影恰好有同一特质，比如说都是动作片，而其他电影则大不相同。你可能喜欢爱情片，而我不喜欢。但是，如果我们不是比较两个人的品味，而是纵览许多用户，喜欢这些电影的用户也喜欢其他一些电影，就可以更加确信这是好的建议。相反，如果仅仅是个巧合，就不会出现任何一致性：喜欢动作片的一些人喜欢爱情片，另一些人不喜欢，这样就不会出现任何模式。

注意，我们也可以从不同的方向看待这个问题。可以从电影的角度观察，比较电影的简介，而不是从用户的角度去观察（两个人是否有类似的品味？）。例如，如果人们对 A 电影和 B 电影的评级相同，根据对 A 电影的想法，就可以合理地猜测对 B 电影的看法，根据预计的兴趣度提出建议。

5.10.1 简单标签推荐系统

让我们将上述思路应用到 StackOverflow 数据集。这个例子有些人为的痕迹，主要的意图是概述问题的解决方法。我们所要做的是从数据集中提取 100 个“测试”用户，根据前 20 个标签的情况，预测他们将对剩下 10 个标签中的哪一个感兴趣。换言之，将他们想象为相对新的用户，我们唯一知道的是他们到目前为止活跃度较高的标签。根据历史，我们能否推荐标签？因为我们也不知道他们在剩下的 10 个标签上的实际表现，所以可以通过比较验证我们的猜测。

我们根据用户简况尝试推荐，方法很简单：取得一个用户，对于已知的 20 个标签，将他（她）的情况与已知用户对比，确定相似度。然后，计算已知用户在剩下 10 个标签上的兴趣度，以他们和目标用户的相似度作为权重，预测目标用户的兴趣度。

如果我们打算用相似度计算加权平均值，需要两样东西。首先，需要随用户简况相似度增加的一个计量手段：如果发现某人与你非常相似，在预测你可能喜爱（或者不喜爱）的事物时，就要比其他人更多地考虑这个人的偏好。其次，需要一个永远为正的相似度值，用于计算加权平均值。

可以使用距离计算相似度吗？经典的欧几里得距离符合一个条件——总是为正——但是不符合另一个条件：两个简档越相似，距离越小。然而，这并不难解决。不使用距离，而

是采用其倒数，就可以得到随距离减小而增大的数量。但是，还有一个小问题：如果两个简档完全相同，距离为 0，倒数会变成无限大。我们可以略做调整，用如下公式计量相似度：

```
Similarity(X,Y) = 1.0 / (1.0 + Distance (X,Y))
```

现在，完全相同的简档相似度为 1（100%），用户相差越大，相似度越小，极限为 0。如果你将推荐系统更多地看作猜测而非科学，那至少在某种程度上是对的。正如所使用的距离或者标度特征的方法没有明显的唯一选择那样，相似度的合适定义也取决于数据。最终，“合适”的选择就是最有效的选择，由交叉验证确定。

5.10.2　实现推荐系统

我们将在新的脚本文件中探索这个问题，以免将两个现有的脚本搞乱。和另外两种方法一样，首先在内存中打开 userprofiles-toptags.txt 文件，提取观测值和表头，过滤只包含零值的观测值。

为了使用户简档可以比较，我们需要处理聚类分析中遇到的相同问题，也就是，用户之间活跃度水平的极大差异。首先重新标度观测值，使最大的标签取值为 100%，最小的为 0%，并提取出 100 个观测值——用于交叉验证的测试目标。

程序清单 5-20　准备数据集

```
let scale (row:float[]) =
    let min = row |> Array.min
    let max = row |> Array.max
    if min = max
    then row
    else
        row |> Array.map (fun x -> (x - min) / (max - min))

let test = observations.[..99] |> Array.map scale
let train = observations.[100..] |> Array.map scale
```

在进入预测部分之前，我们需要几个助手函数。一个是使用距离的相似性函数 similarity。为了方便起见，我们还要编写一个权重函数 weights，该函数以一个浮点数组为参数，重新标度它们，使其总和为 100%，用于加权平均值。split 函数则将前 20 个标签与 10 个“未知”标签分开。

程序清单 5-21　相似性和实用工具函数

```
let distance (row1:float[]) (row2:float[]) =
    (row1,row2)
    ||> Array.map2 (fun x y -> pown (x - y) 2)
    |> Array.sum

let similarity (row1:float[]) (row2:float[]) =
    1. / (1. + distance row1 row2)
```

```
let split (row:float[]) =
    row.[..19],row.[20..]

let weights (values:float[]) =
    let total = values |> Array.sum
    values
    |> Array.map (fun x -> x / total)
```

■ **提示：**||>操作符可以用于一次在两个集合上运算的函数，如 Array. map2 或 Array.zip。用“双管道向前”操作符，两个参数不是单独传递给函数，而是以元组形式提供，这看起来与管道操作更为一致。

现在，我们已经为预测部分的研究做好了准备。在给定的用户历史中，我们将只保留应该知道的部分（前 20 个标签），然后计算训练集中每个用户的相似性。完成之后，我们将取后 10 列（我们将要预测的），对每列计算相似性与已知使用量水平乘积之和，这将生成 10 个数值组成的数组——10 个标签的预测兴趣度。

程序清单 5-22 计算用户预测

```
let predict (row:float[]) =
    let known,unknown = row |> split
    let similarities =
        train
        |> Array.map (fun example ->
            let common, _ = example |> split
            similarity known common)
        |> weights
    [| for i in 20 .. 29 ->
        let column = train |> Array.map (fun x -> x.[i])
        let prediction =
            (similarities,column)
            ||> Array.map2 (fun s v -> s * v)
            |> Array.sum
    prediction |]
```

现在，我们已经做好了预测的准备，下面在第一个测试目标上尝试：

```
let targetTags = headers.[20..]
predict test.[0] |> Array.zip targetTags
>
val it : (string * float) [] =
  [|("objective-c", 0.06037062258); ("php", 0.1239587958);
    ("python", 0.1538872057); ("regex", 0.06373121502); ("ruby", 0.1001880899);
    ("ruby-on-rails", 0.09474917849); ("sql", 0.07666851406);
    ("sql-server", 0.05314781127); ("wpf", 0.02386000125);
    ("xml", 0.03285983829)|]
```

对于 10 个标签中的每一个，我们都得到了从 0%到 100%的预测兴趣度水平。这次，我们的最佳候选是 Python，兴趣度为 15%，之后是 PHP（12%）和 Ruby（10%）。这个预测是否出色？我们可以将其与实际情况对比：

```
> test.[0] |> split |> snd;;
val it : float [] =
  [|0.0; 0.0; 0.03846153846; 1.0; 0.0; 0.0; 0.0; 0.0; 0.0; 0.0|]
```

如果我们根据最大的预测兴趣度（15.4%）推荐一个标签，那么应该推荐第 3 个标签（Python），这确实是一个用户活跃度很高的标签，预测很成功。与此同时，我们遗漏了该用户真正的最爱——Regex，它在预测中兴趣度大约为 6%，不是得分最高的推荐项。

5.10.3　验证做出的推荐

仅仅观察一个例子，无法决定推荐系统是否出色。我们假定以如下方式使用推荐系统：计算每个位置标签的预测兴趣度，提议使用具有最高预测兴趣度的标签。计量推荐是否“好”的方法之一是用户是否真正表现出对推荐标签的兴趣。这是一个不可靠的评估。例如，在前面的例子中我们认为“Python”是一个好的推荐，但是遗漏了更好的推荐“Regex”。与此同时，这个推荐并不差，我们提出了用户可能确实感兴趣的标签。

尽管这可能不是最好的计量手段，但是方便易行，所以我们将继续这么做。我们逐个选取 100 个测试目标，简单地将预测标签与用户活跃度高的标签相匹配的情况视为加分项。我们只需要捕捉感兴趣的 10 个标签的实际活跃度、每个标签的预测兴趣度，捕捉其中最大者，检查推荐标签观测到的活跃度水平是否确实大于 0 即可。出现这种情况时，我们就视为预测成功，并计算正确调用的比例。

程序清单 5-23　正确推荐比例

```
let validation =
    test
    |> Array.map (fun obs ->
        let actual = obs |> split |> snd
        let predicted = obs |> predict
        let recommended, observed =
            Array.zip predicted actual
            |> Array.maxBy fst
        if observed > 0. then 1. else 0.)
    |> Array.average
    |> printfn "Correct calls: %f"
```

根据这一指标，我们得到了 32%的正确推荐率。面对现实吧——这一数字没有说服力。然而，我们实际上并没有可供比较的基准。最简单的预测方法就是预测平均评级最高的标签，忽略用户的相似性。下面我们对该方法进行简易评估。

程序清单 5-24 简单推荐的准确率

```
let averages = [|
    for i in 20 .. 29 ->
        train |> Array.averageBy(fun row -> row.[i]) |]

let baseline =
    test
    |> Array.map (fun obs ->
        let actual = obs |> split |> snd
        let predicted = averages
        let recommended, observed =
            Array.zip predicted actual
            |> Array.maxBy fst
        if observed > 0. then 1. else 0.)
    |> Array.average
    |> printfn "Correct calls: %f"
```

使用最简单的推荐方法，我们的准确率下降到 13%。这并不能说 32%的准确率很高，只是证明基于相似度的方法有一定的可取之处。

我们将维持推荐引擎，不尝试进一步的改进。我相当肯定可以大大改进它，但是不认为能够从此过程中得到更深刻的认识。我将以几个值得考虑的要点作为结束，以帮助你进一步从数据集或者其他方面探索这个问题。

首先，我们的样本不是该方法的最典型用例。在典型的场景下，你知道人们对某些事项的评估，而不知道他们对其他事项的评估；例如，某人看过哪些电影以及对这些电影的评价（从“糟糕”到“出色”）。这和我们的例子有两方面的不同：在上述情况中，我们知道某个事项是否得到评价，所有的评价都有一致的标度。相比之下，我们的数据集对于每个用户的标度差别很大，因为我们计量的是活跃度，没有清晰地区分空值（用户从未看过这个标签）和不感兴趣的标签（用户有机会查看该标签，但是不感兴趣）。因此，比较用户或者标签的相似度有些复杂。

在更为典型的情况下，我们可能面对稍微不同的难题。一行数据中可能大部分是未知的数值（用户尚未评价的所有事项），比较不同用户需要通过公共项进行，比计算两个完整的行之间的距离更复杂。这也指出了协同过滤方法的局限性，而在我们的例子中，这种局限性并不明显。当全新的用户出现在我们的系统中时，没有可用的历史数据，因此也就没有任何可用于生成推荐的依据，这称作“冷启动”问题。如果进一步试验模型，就会或多或少地遇到如下情况：默认情况下，推荐系统预测 Python，这是 10 个目标标签中最流行的标签。只有在我们拥有了强大、定义完整的用户简档之后，推荐引擎才能生成不同的推荐。这是通常可以预料的结果：在信息有限的情况下，引擎将推荐广受欢迎的事项，只能为提供许多信息的用户生成“有趣”的选择。

最后总结一下这个主题：为了生成推荐，我们必须扫描每个用户。这明显是代价很高的操作，隐含着明显的标度问题。我们已经按照特征（而不是按照用户）尝试了该问题的解决，并且尝试检测了 30 个标签（而不是 1600 个用户）之间的相似性，我们甚至可以预先计算列间的相似性，这能够明显地减少计算量。

5.11　我们学到了什么？

在本章中，我们介绍了无监督学习。在有监督学习方法（如分类或者回归）中，算法的目标是从一个训练集中学习，指导对特定问题的回答。相比之下，无监督学习是在“简单”数据中自动找出有趣的特征或者结构，而不需要用户的明确引导。在探索无监督技术的过程中，我们讨论了几个主题，目标是取得一个数据集，了解是否能够不依靠人工研究数据发现有趣的模式，而是应用更为机械的方法，系统地提取和总结数据中的信息。

我们对数据应用了两种互补的技术：k-均值算法聚类分析和主成分分析。两种方法有类似的特质：两者都试图在数据中发现模式和简化信息，根据某种距离的概念聚集类似的元素，将数据集归纳为更浓缩的表现形式。两种方法的不同之处在于寻找的模式类型。k-均值聚类试图将观测值分组为多个聚类，在理想情况下，同一个聚类中的项目相互接近，聚类本身尽可能不同。k-均值确定质心——本质上是“原型”，足以描述一大类观测值。

相比之下，主成分分析的焦点不在观测值本身的相似性上，而是哪些特质同时变化，以及观测值与平均值的差别。通过检查协方差矩阵，该方法识别同时变化的特征，试图提供替代的表现方式，将通常同向或者反向变化的特征集合在一起，并以强调观测值间最大区分因素的方式组织。

在两种方法中，关键的好处之一是数据集更简单、更紧凑的描述。聚类中，我们可以用少数表现一致的原型描述数据集，而无须使用许多独特的观测值。在 PCA 中，我们得到几个新特征——主成分，可以很好地区分观测值，而无须使用多个不相关的特征。结果是，我们可以开始开发和理解比原始数据更有意义、更容易理解的维度，而无须处理难以解读的事实。这些方法使我们能够开发一个词汇表，更好地表述数据，还可能更好地理解数据的“工作”方式和深刻认识它们的方法。

与此同时，这两种技术都在数据中找出特定类型的模式。这种模式是否真正存在无法保证，即使存在，也不能保证对我们有益。我们可能发现没有实际用处的聚类，例如，无法据此采取行动，或者它们仅仅说明了显而易见的现实。

我们讨论的另一方面是距离的概念和考虑标度的重要性。聚类集合近似的观测值，在此框架下，为算法提供有意义的距离是至关重要的。在这个背景下，我们最终需要预先处理观测值，将其归纳为可比较的用户简档。PCA 组合与平均值差异类似的特征，引起了规格化和相关的讨论，这两个概念围绕一个思路：重新标度计量值，使其变化可比较。

最后，我们概述了类似思路的应用——不是从数据集中提取信息，而是用于生成推荐。在 3 个例子中，我们观察了许多个体的表现，发现尽管存在个体差异，整体上仍然存在某些模式。在最后一个例子中，我们从“在许多情况下有类似表现的人，可能在其他情况下也有类似表现”的假设出发，探索了发现相似性并加以应用，确定未知情况下可能表现，从而做出推荐的方法。

第 6 章

树与森林

从不完整的数据做出预测

我的侄女们最喜欢的游戏之一是猜谜游戏。一个人想到某件事物，其他人试图通过询问只有“是”、“否”两个答案的问题，猜出这一事物。如果你以前玩过这个游戏，可能会在游戏中看到如下模式：首先提出消除几大类可能答案的问题，比如“是动物吗？”，然后随着信息的收集逐渐缩小问题的焦点。这比从一开始就询问“你想的是斑马吗？”之类的问题更有效。一方面，如果答案是“是”，你就赢了，因为只有一种可能的答案。另一方面，如果答案是 “否”，那么你就处于相当不利的局面，几乎没有得到任何信息。

决策树的运作方式类似于这种游戏。它们模拟人类询问一系列问题做出诊断的方式，试图快速消除不好的答案，根据目前所了解的所有信息决定下一个问题。在这一背景下，特征可以视为人们询问的关于观测点特性的问题。决策树将从训练集学习、分析，根据此时已知和试图分类的与观测值有关的所有信息，确定哪个特征提供最多的信息。

在本章中，我们将在一个经典的数据集上开展工作，逐步学习越来越强大的分类算法，从决策桩开始，扩展到决策树和森林。在这一过程中，我们将：

- 学习如何构建可处理任何类型数据（包括数据缺失的情况）的灵活决策树。我们还将了解如何使用可区分联合和选项等 F#类型，以极高效的方式建立树形结构模型。
- 引入熵，作为计量数据集包含信息量的手段，并用其比较特征，决定选择哪一个。
- 讨论过度拟合的风险以及解决方法。特别是，我们将探索更高级的交叉验证方法，如 k-折方法，以及相同的思路如何引出类似方法——将简单预测程序聚合为健全程序的通用方法。

6.1 我们所面临的挑战：“泰坦尼克”上的生死存亡

我们将在本章中使用一个经典的数据集，目标是使用“泰坦尼克”号的乘客名单以及每

个人的一些人口学信息，预测他们的命运。数据集和问题都很有趣，原因有三。首先，略过令人毛骨悚然的主题，这个问题是一大类问题中的典型。“那位访问者会不会点击网站上的这个链接？”或者“哪些客户会选择小、中或者大的订单？”本质上都是相同的问题：使用个人的有关信息和过去的表现，试图构建一个分类模型，从有限的可能中预测一种结果。

其次，数据本身在现实世界中也是很典型的——十分凌乱。它包含了不同类型特征的组合，从数值（乘客花了多少钱买票？）到分类（乘客是男性还是女性？）以及两者之间的所有种类（乘客姓名和头衔）。还有一些数据缺失。理想状况下，我们希望分类程序足够灵活，能够处理所有的变量，而不会造成太多麻烦。

最后，由于显而易见的历史原因，数据本身也很有趣。“泰坦尼克”号的浩劫在很长时间内都深深吸引着人们，但很少有人为发生在一个世纪之前的偶然事故建立一个实用的预测性模型。这个数据集提供了一个机会，看看分析事故数据能够认识到什么。

6.1.1　了解数据集

我们使用的数据集来自 Kaggle 的“泰坦尼克：灾难中的机器学习”竞赛：https://www.kaggle.om/c/titanic-gettingStarted。网上很容易找到变种（例如，http://www.encyclopedia-titanica.org）。但是，我们认为使用 Kaggle 的参考数据集应该很有趣，这样，你可以尝试自己的技能，看看自己所能取得的进展。为方便起见，我们将同一个数据集上传到 OneDrive：http://1drv.ms/1HUh1ny。

数据集是一个 CSV 文件 titanic.csv，组织得很好，有一个描述 12 列的表头。当然，我们可以编写一些代码解析它，但是为何不使用 CSV 类型提供程序呢？创建一个包含 F#项目 Titanic 的解决方案，将 titanic.csv 文件添加到项目中，并添加 fsharp.data NuGet 包。

现在，我们已经做好准备，可以开始探索数据了。在 script.fsx 文件中，添加对 FSharp.Data 的引用（根据包的当前最新版本，路径的细节可能不同），查看样本数据、创建一个 Titanic 类型，并添加一个类型别名 Passenger，以便用更直观的方式引用数据集的每一行。现在，读入数据集：

程序清单 6-1　使用 CSV 类型提供程序读取 Titanic 数据集

```
#r @".\packages\FSharp.Data.2.2.2\lib\net40\FSharp.Data.dll"
open FSharp.Data

type Titanic = CsvProvider<"titanic.csv">
type Passenger = Titanic.Row

let dataset = Titanic.GetSample ()
```

CSV 类型提供程序为我们创建了一个很好的类型，包含所有属性，为我们探索数据提供了类型的所有好处。现在，我们可以开始工作了。如果目标是预测人们幸存或者遇难，第一步是获得该值的基准。首先计算每类乘客的人数，然后在脚本中添加如下代码并在 F# Interactive 中运行，算出单个乘客的生存概率：

```
dataset.Rows
|> Seq.countBy (fun passenger -> passenger.Survived)
|> Seq.iter (printfn "%A")
dataset.Rows
|> Seq.averageBy (fun passenger ->
   if passenger.Survived then 1.0 else 0.0)
|> printfn "Chances of survival: %.3f"

>
(false, 549)
(true, 342)
Chances of survival: 0.384
```

我们有 891 位乘客，其中只有 342 人幸存。换言之，在不考虑其他信息的情况下，每个乘客只有 38.4%的幸存概率（或者说，有 61.6%的概率无法弃船逃生）。如果我们想要改进预测，就需要找出帮助我们识别幸存概率超过 50%的乘客的特征。其他特征尽管有很大的信息量，但是不会直接为预测带来实质性的差异，因为它们不会改变默认的决策。

6.1.2 观察各个特征

有了上述框架，我们就可以开始观察各个特征了。我们所寻求的是快速地将乘客分为不同组，然后计算每组幸存率的方法。计算幸存率很简单：给定乘客样本，简单地计算幸存乘客与样本中乘客总数的比率。接着，可以根据不同的条件分组乘客——比如性别或者客舱等级——并显示每一组的幸存率。

程序清单 6-2 计算不同分组的幸存率

```
let survivalRate (passengers:Passenger seq) =
    let total = passengers |> Seq.length
    let survivors =
        passengers
        |> Seq.filter (fun p -> p.Survived)
        |> Seq.length
    100.0 * (float survivors / float total)

let bySex =
    dataset.Rows
    |> Seq.groupBy(fun p -> p.Sex)

bySex
|> Seq.iter (fun (s,g) ->
    printfn "Sex %A: %f" s (survivalRate g))
```

```
let byClass =
    dataset.Rows
    |> Seq.groupBy (fun p -> p.Pclass)
byClass
|> Seq.iter (fun (s,g) ->
    printfn "Class %A: %f" s (survivalRate g))
>
Sex "male": 18.890815
Sex "female": 74.203822
Class 3: 24.236253
Class 1: 62.962963
Class 2: 47.282609
```

很明显，这两个特征都有很大的信息量。事实证明，“泰坦尼克”号上的女性很幸运，有74%的幸存概率。类似地，头等舱乘客的幸存概率相当高（63%），而可怜的三等舱乘客们的前景要可怕得多，只有24%的幸存概率。套用乔治•奥威尔的名言，“所有乘客都是平等的，但是有些乘客实际上高人一等。”这有一些相当显而易见的社会学解释，但是我将把它们留给读者：毕竟，数据只说明事实，而没有说明原因。

6.1.3 构造决策桩

我们已经有了足够的信息实现一个极其简单的分类器，以及训练它的规程。取得样本数据，按照上面的做法将其分成不同组，对每一组预测最频繁出现的标签。这类模型称作“决策桩”，是你所能构建的最短决策树。决策桩是树的组成部分，是最小化的分类器，常常作为其他模型的组成部分。

举个例子，我们如何根据乘客的性别构建决策桩呢？首先，从样本中取得所有乘客，并根据选中的特征（本例中是性别，“男”或“女”）分解为不同的组。然后，对每一组寻找选中的标签值（本例中是幸存与否），确定每一组中最频繁出现的值。接着，我们可以构建一个分类器（桩）：以乘客为输入，计算特征值，在组中查找，返回各组中最常见的标签（见图6-1）。

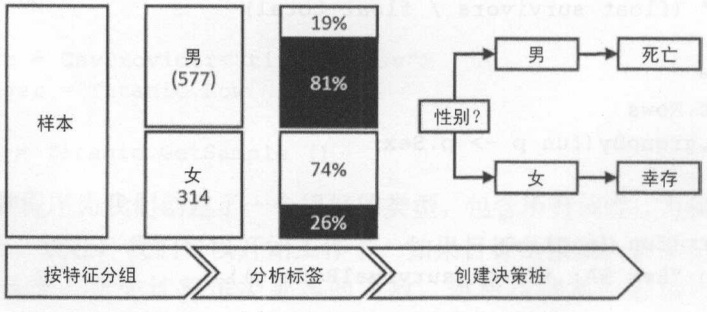


图6-1 以性别为特征，构建一个决策桩

■ 注意：在前面几章中，我们使用术语"分类"描述分类器的预测值。这里，我们将用"标签"代替，这也是常见的术语。在我们的例子中使用它很方便，因为客户的"分类"（Class）是数据集中的一个特征（表示客舱等级），对两者使用相同术语可能引起歧义。

如果我们将特征和标签变成从观测值中提取值的函数，这一过程就可以完全通用。假定采用一个泛型类型'a 的序列代替乘客，某个特征就是这样一个函数：给定'a，返回值'b，同样，标签就是给定'a，返回另一个类型'c。在我们的特殊例子中，'a 是一个乘客，'b 是一个字符串（取值为"male"（男）或者"female"（女）），c'是一个布尔值（幸存或者死亡）。类型中唯一的附加要求是，'b 和'c 必须支持相等性，因为我们需要同时按照特征和标签值进行分组和计数。

那么，我们的实现是什么样的？和前面描述的很相似：

程序清单 6-3 学习一个决策桩

```
let mostFrequentLabelIn group =
    group
    |> Seq.countBy snd
    |> Seq.maxBy snd
    |> fst

let learn sample extractFeature extractLabel =
    // group together observations that have the
    // same value for the selected feature, and
    // find the most frequent label by group.
    let groups =
        sample
        |> Seq.map (fun obs -> extractFeature obs, extractLabel obs)
        |> Seq.groupBy fst
        |> Seq.map (fun (feat,group) -> feat, mostFrequentLabelIn group)
    // for an observation, find the group with
    // matching feature value, and predict the
    // most frequent label for that group.
    let classifier obs =
        let featureValue = extractFeature obs
        groups
        |> Seq.find (fun (f,_) -> f = featureValue)
        |> snd
    classifier
```

注意，程序清单 6-3 主要是为了说明，因此我没有在两个函数上加入任何类型注释。如果在 F# Interactive 中运行这段代码，或者简单地将鼠标悬停于 learn 的定义上，就会看到如下内容：

```
val learn :
  sample:seq<'a> ->
    extractFeature:('a -> 'b) -> extractLabel:('a -> 'c) -> ('a -> 'c)
    when 'b : equality and 'c : equality
```

这有一点可怕，因此让我们做些解释。上述代码说明，learn 是一个函数，给定观测值的一个样本（'a 的序列），从观测值'a 提取特征值'b 的函数（'a -> 'b）以及从观测值'a 提取标签值'c 的函数（'a -> 'c），你将得到一个函数'a -> 'c——以观测值为参数，返回标签的函数，这正是分类器的预期功能。如果这个函数签名的所有细节不清晰，不用烦恼，这些细节并不很重要。我介绍这些代码的目的是说明 F#类型推理系统的威力。只需描述对输入进行的处理，不需要太多规范，F#就能正确地用泛型类型创建一个函数，甚至推导出我们需要同时支持提取的标签和特征值的相等性。

6.1.4　训练决策桩

如何训练决策桩？我们必须创建标签和特征所使用的规范，将它们定义为适合数据集的函数。调用 learn 将创建一个完善的分类器函数，可以立即运行和评估（暂时省略交叉验证）。

程序清单 6-4　训练和评估决策桩

```
let survived (p:Passenger) = p.Survived
let sex (p:Passenger) = p.Sex

let sexClassifier = survived |> learn (dataset.Rows) sex

printfn "Stump: classify based on passenger sex."
dataset.Rows
|> Seq.averageBy (fun p ->
    if p.Survived = sexClassifier p then 1.0 else 0.0)

>
Stump: classify based on passenger sex.
val it : float = 0.7867564534
```

在整个样本上运行上述代码，可以得到 78.6%的正确预测，显然高于最初的 61.6%，同样，可以轻松地创建一个基于乘客客舱等级的模型：

程序清单 6-5　有多个分支的决策桩

```
let classClassifier = survived |> learn (dataset.Rows) (fun p -> p.Pclass)

dataset.Rows
|> Seq.averageBy (fun p ->
    if p.Survived = classClassifier p then 1.0 else 0.0)
```

正如预期，正确率为 67.9%，比起不采用任何模型来说是一个改进，但是不及前一个例子中所观察到的结果。但是，这一例子的目的并非真的是改进性能。它说明了两个有趣的事实。首先，和第一个例子不同，这个决策桩有三个分支而非两个。决策桩（和后面介绍的决策树）可以容纳超过两个选择。其次，要注意两个特征的类型不同：性别返回的是一个字符串，而客舱等级返回的是正数。类型推理系统为我们自动处理这一差异。

6.2 不适合的特征

显然，仅使用一个特征进行预测的简单模型只能走这么远，此时有一个明显的问题，“如何将多个特征组合为单一模型？”这就是接下来的方向——尝试将更多数据及信息组合起来，将多个决策桩组合形成一棵决策树，改进我们的预测。但是，在研究更复杂的模型之前，必须解决几个问题，决策桩方法相当灵活，但是有一些局限性。

6.2.1 数值该如何处理？

在我们的特征中，考虑这一个：票价（fare）。毕竟，如果乘客的客舱等级能够提供丰富的信息，他们购票的价格也很有可能提供信息。让我们计算不同票价所对应的幸存率，了解它所能提供的信息。

程序清单 6-6 具有连续数字值的特征

```
let survivalByPricePaid =
    dataset.Rows
    |> Seq.groupBy (fun p -> p.Fare)
    |> Seq.iter (fun (price,passengers) ->
        printfn "%6.2F: %6.2f" price (survivalRate passengers))
>
  7.25:   7.69
 71.28: 100.00
  7.93:  44.44
 53.10:  60.00
// snipped for brevity...
  5.00:   0.00
  9.85:   0.00
 10.52:   0.00
```

得出的数字没有问题，但是这似乎不是处理这一特征的好办法。特征分为 248 种不同的个例，大部分的幸存率是 0%或者 100%。根据这一模型，花费 39.4 元购买船票的乘客有 100%的幸存率，花费 39.6 元的只有 50%，39.6875 元的则为 0%。这明显是很愚蠢的预测。我们的问题在于，票价和客舱等级的特性不同。客舱等级是离散的状态（头等舱、二等舱或者三等舱；没有其他选择），而票价理论上可能取无穷多个值。这两个特征的另一处不同在于，票价表示的是一个真实的数字。例如，可以将两个票价相加或者相减，而乘客客舱等级之和没有任何意义。

问题是我们的决策桩只能处理分类特征。如何包含连续数据？方法之一是将这个问题简化为已知的问题，将票价转换为一个分类特征。例如，我们可以创建一个新特征“票价等级”（Fare Level），该特征有两种级别，根据其低于或者高于平均值定义为“便宜”（Cheap）或者“昂贵”（Expensive）。

程序清单 6-7　将票价转换为离散的范围

```
let averageFare =
    dataset.Rows
    |> Seq.averageBy (fun p -> p.Fare)

let fareLevel (p:Passenger) =
    if p.Fare < averageFare
    then "Cheap"
    else "Expensive"

printfn "Stump: classify based on fare level."
let fareClassifier = survived |> learn (dataset.Rows) fareLevel

dataset.Rows
|> Seq.averageBy (fun p ->
    if p.Survived = fareClassifier p then 1.0 else 0.0)

>
val it : float = 0.6621773288
```

结果不是很好，但是重点不在这里：现在我们有了将连续特征转换为离散特征的一种方法。这种方法称作“**离散化**”，也是非常通用的方法：将特征分解为任何数量的连续区间，并将观测值分配到一个“箱子”（Bin）中。

我们解决了一个问题，但是也发现了另一个问题：在许多离散化连续特征的方法中，应该选择哪一个？该问题归结为如下问题：给定两个特征（同一个底层特征的两个离散化范围），应该选择哪一个以得到最多信息？下一小节，我们解决另一个问题——缺失数据。

6.2.2　缺失数据怎么办？

现在考虑另一个可用特征。假定我们想使用 Embarked（每位乘客的出发港口）作为模型中的一个特征，首先观察按照出发港口计算的幸存率。

程序清单 6-8　具有漏失值的特征

```
let survivalByPortOfOrigin =
    dataset.Rows
    |> Seq.groupBy (fun p -> p.Embarked)
    |> Seq.iter (fun (port,passengers) ->
        printfn "%s: %f" port (survivalRate passengers))

>
S: 33.695652
C: 55.357143
Q: 38.961039
: 100.000000
```

显然，在法国瑟堡（C）登船的幸存概率略高于昆士敦（Q）或者南安普顿（S）。但是，有100%幸存率的那个无名分类是什么？

这就是缺失数据。对于每个乘客，我们都知道其性别和客舱等级，但是对于某些乘客，并不知道他们是从哪里出发的。这个问题是真实存在且非常常见的。遗憾的是，在现实生活中，大部分数据集都有一些问题，漏失值是最可能出现的一个。那么，我们该怎么做呢？有诱惑力的选择之一是淘汰不完整的记录。但是，你可能最终抛弃了许多有价值的信息，特别是在数据集中有多个特征缺失数据的情况下。

但是，我们不能将空字符串当成有效的出发港口！至少，应该清晰地将其标记为缺失数据，以便决定之后的处理方法。可以创建一个专门的可区分联合，但是在有选项（Option）类型可用的情况下这明显太过分了。选项类型有两种值，<'T>或者None。

在这个框架下，选项（对不起，这是个双关语）略微地改变了特征的定义方式，要求通过返回一个Option明确地定义漏失值，None表示缺失数据。结果是，学习函数需要明确处理这种情况的策略。程序清单6-9中的代码片段说明了一种处理方法。与前面的版本相比，主要的变化是从样本中过滤具有漏失值的例子，然后仅为这些情况创建一个分支，将其保存为一个Map类型值以提供方便。在存在漏失值的情况下，我们将简单地预测样本中最常见的标签。

程序清单6-9 加入漏失值

```
let hasData extractFeature = extractFeature >> Option.isSome

let betterLearn sample extractFeature extractLabel =
    let branches =
        sample
        |> Seq.filter (extractFeature |> hasData)
        |> Seq.map (fun obs -> extractFeature obs |> Option.get, extractLabel obs)
        |> Seq.groupBy fst
        |> Seq.map (fun (feat,group) -> feat, mostFrequentLabelIn group)
        |> Map.ofSeq
    let labelForMissingValues =
        sample
        |> Seq.countBy extractLabel
        |> Seq.maxBy snd
        |> fst
    let classifier obs =
        let featureValue = extractFeature obs
        match featureValue with
        | None -> labelForMissingValues
        | Some(value) ->
            match (branches.TryFind value) with
            | None -> labelForMissingValues
            | Some(predictedLabel) -> predictedLabel
    classifier

let port (p:Passenger) =
```

```
        if p.Embarked = "" then None
        else Some(p.Embarked)

let updatedClassifier = survived |> betterLearn (dataset.Rows) port
```

注意：Option 类型上的模式匹配结果是，classifier 函数中的逻辑流向很容易跟踪，但是，过滤具有漏失值的例子，然后提取真实值不是巧妙的做法。Seq 模块包含函数 Seq.choose，可以一次性完成这两件工作，但是要求原始序列包含选项。很遗憾，我们的例子中不是这种情况。

6.3　计量数据中的信息

现在，我们已经有了一种使用单一特征创建分类器，而不管其类型或者是否有漏失值的方法。这种方法很好，但是我们不能限制自己一次仅使用一个特征，忽略其他可用特征中存在的信息。

我们将按照与“20 个问题游戏”相同的思路扩展模型，交替推进，在每一轮次中，你可以从一系列特征中询问一个关于观测值的问题：乘客是男性还是女性？他（她）乘坐的是头等舱、二等舱还是三等舱？根据这些问题的答案，你必须做出决定：是否有足够的信息做出预测（幸存与否），或者，根据得到的答案（“乘客是男性”），你是否希望提出另一个可能增大得出正确答案概率的问题？

关键问题是，如何计量和比较信息，以便决定哪一序列问题最佳。这是下一节将要讨论的主题。

6.3.1　用熵计量不确定性

特征本身并不能提供很多信息；信息在你可能得到的答案中，以及这些答案对决定哪个标签最可能出现的帮助上。为了计量问题提供的信息量，我们需要考虑哪一个答案更好，是知道“乘客是男性”更好，还是“乘客坐的是头等舱”更好？

现在，我们真正关心的不是乘客是男是女，而是如果我告诉你乘客是男性，你对他是否幸存有多大的确定性？我能告诉你的最糟糕的事情是，“如果乘客是男性，他的幸存概率是 50%。”这实际上说明该信息毫无价值。相反，如果男性的幸存概率为 100%——或者相反——你的工作就完成了：这是完美的信息，没有任何附加问题能够改进它。

直觉是，一个群体的分布越均匀，我们所拥有的信息就越少：任何一种猜测都不比其他猜测好。信息理论为此定义了一个指标：熵（具体地说，是香农熵）。如果有一个样本群体，其熵的计算如下：

```
entropy(sample) = sum [ - p(x) * log p(x) ]
```

其中，$p(x)$是样本中 x 的比例，也就是观察到一个值 x 的概率。

■ **注意**：对数底数的选择并不重要，只要保持一致即可。不同的底数造成不同的单位（比特，奈特等），但是不影响样本的排序。因此，我们使用自然对数，它在 F#中最容易使用。

我们在 F#中实现熵的计算，以更好地观察其工作原理。给定一个数据样本，我们将计数每个类别中的元素数量，除以总数，计算 $p \log p$。注意，$x \log(x)$在 $x=0$ 时没有定义，我们将以右极限 0 代替，最终结果如下：

程序清单 6-10　样本的香农熵

```
let entropy data =
   let size = data |> Seq.length
   data
   |> Seq.countBy id
   |> Seq.map (fun (_,count) -> float count / float size)
   |> Seq.sumBy (fun f -> if f > 0. then - f * log f else 0.)
```

■ **提示**：F#有一个内建函数 id，该函数将输入值作为输出返回，这对程序清单 6-10 中的情况很有用。在该程序中我们希望找到序列中的特有元素，计算每个元素的数量。

在 F# Interactive 中运行两个例子，快速探索熵的表现：

```
> [1;1;1;2;2;2] |> entropy;;
val it : float = 0.6931471806
> [1;1;1;1;1;1] |> entropy;;
val it : float = 0.0
```

样本的信息量最小时（每个标签出现的次数相等），熵最大；对于组织完善（可预测）的样本，熵将降为 0。图 6-2 对比了两个特性大不相同的例子，说明熵与我们直觉中的组织（或者纯度）概念的对应关系：样本的纯度越高，就越容易猜测，熵也越小。顺便提一句，熵也能够处理超过 2 个标签，因此它是一个方便的指标。

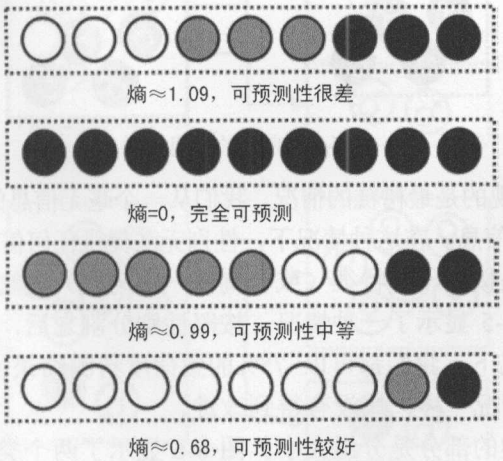


图 6-2　比较样本信息和熵

6.3.2　信息增益

香农熵为我们提供了计量样本信息可用性的手段——熵越低，信息越好。这很有用，但并不是我们所追求的。我们想要的是：如果有一个样本，需要选择两个问题（特征）中的一个，应该选择哪一个？

为了做出这个决策，我们将使用信息增益的概念，从其数学表达开始，然后讨论它的意义所在。假定你有一个样本，通过观测特征 F，可以将样本分成 n 个组：sample(1), ..., sample(n)。特征 F 的熵增益为

```
gain = entropy(sample) - (proba(sample(1))*entropy(sample(1))+...proba(sample(n)) *
entropy(sample(n)))
```

这个公式有何意义呢？第一部分 entropy(sample)表示的就是当前的熵——也就是原始样本信息的质量。你可能已经发现，第二部分是一个平均值——根据特征分隔样本得到的平均熵。在深入实现这一公式之前，我们先用几个图形示例说明思路。在每个图像中，我们将思考一个虚构的乘客样本，其中幸存者被显示为白色，其余情况显示为黑色，乘客的性别用经典的火星和金星符号表示。

在图 6-3 中，我们看到了理想的情况，从一个很不适合预测的样本开始（幸存者和遇难者各一半）。如果按照性别分割样本，就会得到两个完美的分组，一个完全是幸存者，另一个全是死难者。在这种情况下，性别就是完美的特征了。如果我们询问乘客的性别，就能准确知道其命运——此时熵增益最高。

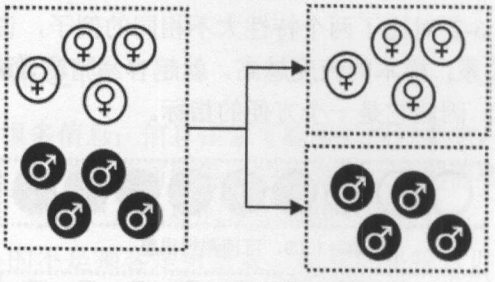


图 6-3　完美的信息增益

相比之下，图 6-4 表现的是最糟糕的情况。我们从一个毫无信息的样本开始，按照性别分割之后，两个样本仍然毫无信息。在这种情况下，性别无法提供任何信息，对应的信息增益为 0。

注意，熵增益指的是按照一个特征（或者询问一个问题）分割样本所得到的信息量，而不是熵本身。例如，图 6-5 显示了一种情况，按照性别分割之后，我们得到了两个有一定信息量的样本（在两种情况下，我们都可以 75%的置信度猜测结果），但是和原来的情况相比没有改善，正如前面的情况一样，熵没有得到改进。

熵增益中另一个有趣的部分是分组大小。图 6-6 显示了两个类似但是稍有不同的情况。在两种情况下，按照性别分割样本都提供了更多信息。但是，下方的情况好于上方。在上方

的情况下，如果我们知道乘客是男性，就拥有了完美的信息，但是只有25%的概率听到这个回答。相比之下，在第二种情况下，我们有75%的概率知道乘客是女性，并获得完美的信息。使用相同的特征，我们在一种情况下比另一种情况更快地完成预测，这种特征从信息增益的构造方式中反映出来，因为公式考虑了分组的增益，并以每个组的大小作为权重。

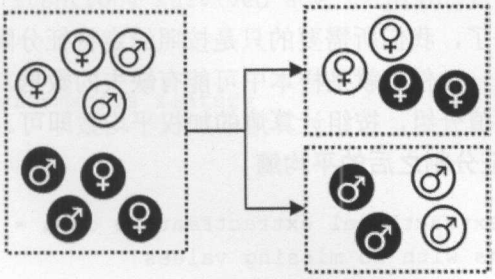


图 6-4　信息增益为 0

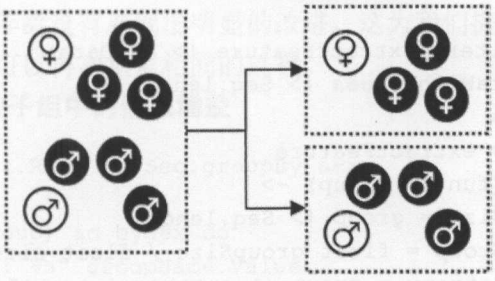


图 6-5　好的信息，但是没有增益

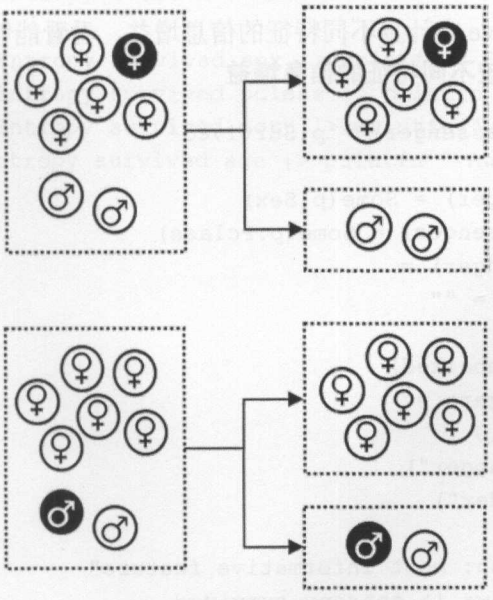


图 6-6　通过分组获得的信息

换言之，熵增益为我们提供了一个指标，如果我们希望高效地研究标签，且一次只能问一个问题，就可以通过这个指标比较最有优势的特征。

6.3.3　实现最佳特征识别

第一部分已经准备好了，我们所需要的只是按照给定特征分隔样本时的平均熵。这相当简单，因为需要注意的唯一一件事就是样本中可能有缺失的数据。我们只需要过滤该特征没有值的观测点，按照特征值分组，按组计算熵的加权平均数即可。

程序清单 6-11　特征分割之后的平均熵

```
let splitEntropy extractLabel extractFeature data =
    // observations with no missing values
    // for the selected feature
    let dataWithValues =
        data
        |> Seq.filter (extractFeature |> hasData)
    let size = dataWithValues |> Seq.length
    dataWithValues
    |> Seq.groupBy extractFeature
    |> Seq.sumBy (fun (_,group) ->
        let groupSize = group |> Seq.length
        let probaGroup = float groupSize / float size
        let groupEntropy = group |> entropy extractLabel
        probaGroup * groupEntropy)
```

我们将在 F# interactive 中计算不同特征的信息增益，看看能够发现什么。

程序清单 6-12　比较不同特征的信息增益

```
let survived (p:Passenger) = p.Survived

let sex (p:Passenger) = Some(p.Sex)
let pclass (p:Passenger) = Some(p.Pclass)
let port (p:Passenger) =
    if p.Embarked = ""
    then None
    else Some(p.Embarked)
let age (p:Passenger) =
    if p.Age < 12.0
    then Some("Younger")
    else Some("Older")

printfn "Comparison: most informative feature"
let h = dataset.Rows |> entropy survived
```

```
printfn "Base entropy %.3f" h

dataset.Rows |> splitEntropy survived sex |> printfn " Sex: %.3f"
dataset.Rows |> splitEntropy survived pclass |> printfn " Class: %.3f"
dataset.Rows |> splitEntropy survived port |> printfn " Port: %.3f"
dataset.Rows |> splitEntropy survived age |> printfn " Age: %.3f"

>
Comparison: most informative feature
Base entropy 0.666
  Sex: 0.515
  Class: 0.608
  Port: 0.649
  Age: 0.660
```

性别是明显的胜者，该特征有最低的熵值，因此是信息量最大的特征；客舱等级也有一些潜力，但是出发港口和年龄没有展现出明显的改进。这为我们提供了一个决策桩和两个分组——男性和女性。我们可以再次重复相同的过程。

程序清单 6-13　比较子组中的信息增益

```
let bySex = dataset.Rows |> Seq.groupBy sex

for (groupName, group) in bySex do
    printfn "Group: %s" groupName.Value
    let h = group |> entropy survived
    printfn "Base entropy %.3f" h

    group |> splitEntropy survived sex |> printfn " Sex: %.3f"
    group |> splitEntropy survived pclass |> printfn " Class: %.3f"
    group |> splitEntropy survived port |> printfn " Port: %.3f"
    group | splitEntropy survived age |> printfn " Age: %.3f"
>
Group: male
Base entropy 0.485
  Sex: 0.485
  Class: 0.459
  Port: 0.474
  Age: 0.462
Group: female
Base entropy 0.571
  Sex: 0.571
  Class: 0.427
  Port: 0.555
  Age: 0.565
```

运行上述代码将首先看到，再次在性别上分割产生的增益为 0，这是令人欣慰的（我们已经使用了该信息）。你还将看到，每组特征的顺序不同——在本例中，两个分组中最佳的仍然都是客舱等级，但是每个分支最终会得到不同的特征。最后，还要注意，我们在一次分割之后总是得到至少相同的熵值水平，这意味着，如果添加更多特征并且不断重复进行，结果不会更糟，但可能会将样本分成越来越小的组，而不管这种分割是否真的有意义。所以，我们必须思考一种停止规则，以确定何时进一步分解没有价值。

从编码的角度看，你可能已经想过，上述的代码急需某种重构。人工编写每个特征非常乏味，定义一个特征列表显然更加令人满意。遗憾的是，这里有一些问题。例如，如果考虑性别（sex）和客舱等级（pclass）这两个特征，就会注意到它们的类型不同。性别产生一个字符串选项，而客舱等级返回一个整数型选项，因此，无法将它们包含在同一个列表中。

那么，我们该怎么办呢？有几种选择，但是都不出色。在此我们将照顾简洁性，强制各个特征产生一个字符串选项。与此同时，为特征取个名字也是很方便的。如果我们为特征定义一个类型（如 Feature<'a> = string * 'a -> string option），就可以用类似的方式处理它们，并以更加清晰的方式分析它们。

程序清单 6-14　分析一系列特征

```
let survived (p:Passenger) = p.Survived

let sex (p:Passenger) = Some(p.Sex)
let pclass (p:Passenger) = Some(p.Pclass |> string)

let features =
    [ "Sex", sex
      "Class", pclass ]

features
|> List.map (fun (name, feat) ->
    dataset.Rows
    |> splitEntropy survived feat |> printfn "%s: %.3f" name)
 >
Sex: 0.515
Class: 0.608
```

6.3.4　使用熵离散化数值型特征

作为本方法可能派上用场的一个特例，我们回到前面讨论过的一个问题——如何将连续特征（如年龄或者票价）归纳为一组离散的情况。

在前一小节中，我们为年龄创建了一个特征，将小于 12 岁的乘客称作“年轻人”（Younger），其他人称为“年长者”（Older）。为什么选择 12，而不选择 5、42 或者其他任意值？这个数值的选择完全是随机的，可能有效，但是并不很科学，也不是数据驱动的。

我们可以尝试用熵来解决这个问题，而不是依赖直觉。我们所寻求的是信息量尽可能丰富的年龄阈值。用熵理论的说法就是，我们想要一个最大化熵增益的值。这并不很复杂：只需要从数据集中取得所有可能的年龄值，为每个可能的截止值创建一个特征，找出获得最大信息增益的一个：

```
let ages = dataset.Rows |> Seq.map (fun p -> p.Age) |> Seq.distinct
let best =
    ages
    |> Seq.minBy (fun age ->
        let age (p:Passenger) =
            if p.Age < age then Some("Younger") else Some("Older")
        dataset.Rows |> splitEntropy survived age)
printfn "Best age split"
printfn "Age: %.3f" best

Best age split
Age: 7.000
```

上述方法并不完美。连续变量的归类（“装箱”）是一种神秘的艺术，没有通行的方法。上述方法的好处是完全自动化，因此，可以在没有人工干预的情况下从数据中产生决策桩。

■ **注意**：这个方法的局限性之一是，它假定将数值离散化为两个分类是种好的方法，这并不一定正确。解决该问题的方法之一是反复应用相同规程，检查将识别到的范围进一步分割是否更加有利。但是，这可能很难，因为随着范围变小，造成人为分类的风险越来越大。我们的介绍到此为止，对进一步探索这个问题感兴趣的读者可以使用搜索词“最小描述长度”（minimum description length）。

6.4 从数据中培育一棵决策树

现在，我们已经有了构建决策树所需的两个关键要素。依靠决策桩，我们有了最小的单位，可以经过训练，根据特征处理不同分支中的观测值。凭借熵，可以计量给定观测值样本中某个特征的实用性。现在，我们需要做的就是将决策桩组合成一棵完整的树。从一组特征和一个训练集开始，我们将识别出信息量最大的特征，为每个结果创建一个桩，并对每个分支重复这一操作。

我们将要编写的代码相当通用，因此，按照和第 2 章中类似的方式，将这些代码提取到单独的模块中，我们将在解决方案中添加一个文件（如 Tree.fs），将一些通用的代码移到那里，并在脚本中使用该文件，脚本的焦点将放在为“泰坦尼克”数据集建立模型的特定问题上。

6.4.1 建立树的模型

我们从最终结果——树开始。决策桩和完整的树之间的主要差别是，桩的每个分支都给

出一个明确的最终答案，而树更加复杂：它可能提供一个最终的答案，也可能继续下去，将我们引到另一个桩，提出后续的问题。

大部分时候，当你听到涉及“或”语句的领域描述（“可能是这个或者那个”）时，F#解决方案将包含一个可区分联合，这在我们的例子中非常有效：树是一个答案，或者引出另一棵树的桩。程序清单6-15很直接地反映了我们对问题的描述。

程序清单6-15 以可区分联合的形式建立决策树模型

```
namespace Titanic

module Tree =

    type Feature<'a> = 'a -> string Option
    type NamedFeature<'a> = string * Feature<'a>
    type Label<'a,'b when 'b:equality> = 'a -> 'b

    type Tree<'a,'b when 'b:equality> =
        | Answer of 'b
        | Stump of NamedFeature<'a> * string * Map<string,Tree<'a,'b>>
```

在此有几点简要说明：首先，“泰坦尼克”号的例子中标签是字符串，情况并不一定总是如此。例如，在第1章中，不同的分类用整数表现。我们用泛型表示标签，使树可以处理更广泛的场合。其次，树的数据结构有些复杂——我们将做进一步的解释。前面说过，树产生两个结果之一：得出一个答案（最后一片叶子，包含标签值'b），或者导向一个桩。在这种情况下，我们必须知道3件事：桩用哪个特征构建（NamedFeature<'a>）、特征值缺失时使用哪个值（一个字符串），对于特征的每个可能值（桩的“分支”），下一步是什么——答案还是另一个桩。

如何利用这一结构做出决策？我们想要做到的是：给定一位乘客和一棵树，如果树产生的结果是答案，工作完成。否则，使用与桩相连的特征得出乘客的另一个特征值，继续在对应的树中搜索。如果值为空（None），说明遇到了缺失数据，将使用桩中指定的默认值。我们还决定，为了避免遇到未知的值，将其也当作默认值处理。你可能认为在那种情况下，树应该抛出异常。这样做完全没有问题，但是需要承担潜在问题的代价。

程序清单6-16 使用树做出决策

```
let rec decide tree observation =
    match tree with
    | Answer(labelValue) -> labelValue
    | Stump((featureName,feature),valueWhenMissing,branches) ->
        let featureValue = feature observation
        let usedValue =
            match featureValue with
            | None -> valueWhenMissing
            | Some(value) ->
```

```
        match (branches.TryFind value) with
        | None -> valueWhenMissing
        | Some(_) -> value
    let nextLevelTree = branches.[usedValue]
    decide nextLevelTree observation
```

6.4.2 构建决策树

我们几乎已经完工了！最后一件事是用样本和一组初始特征构建决策树。我们将简单地找出样本中信息量最大的特征，为它创建一个桩，并且持续下去直到没有剩下任何特征，或者剩下少量的数据。

程序清单 6-17 完成的就是这些工作，代码有点冗长，但是并不特别复杂。有两个难点值得一提。首先，特征以 Map<string,Feature>的形式传递。一方面，这样按照名称访问特征和添加/删除特征很容易。另一方面，将映射当成序列使用时，元素以键-值对的形式出现。而且，我们不打算在桩中为漏失值建立一个分支：它们将按照默认值处理。

程序清单 6-17 从样本和特征中发展一棵决策树

```
let mostFrequentBy f sample =
    sample
    |> Seq.map f
    |> Seq.countBy id
    |> Seq.maxBy snd
    |> fst

let rec growTree sample label features =

    if (Map.isEmpty features)
    // we have no feature left to split on:
    // our prediction is the most frequent
    // label in the dataset
    then sample |> mostFrequentBy label |> Answer
    else
        // from the named features we have available,
        // identify the one with largest entropy gain.
        let (bestName,bestFeature) =
            features
            |> Seq.minBy (fun kv ->
                splitEntropy label kv.Value sample)
            |> (fun kv -> kv.Key, kv.Value)
        // create a group for each of the values the
        // feature takes, eliminating the cases where
        // the value is missing.
        let branches =
            sample
```

```
            |> Seq.groupBy bestFeature
            |> Seq.filter (fun (value,group) -> value.IsSome)
            |> Seq.map (fun (value,group) -> value.Value,group)
        // find the most frequent value for the feature;
        // we'll use it as a default replacement for missing values.

    let defaultValue =
        branches
        |> Seq.maxBy (fun (value,group) ->
            group |> Seq.length)
        |> fst
    // remove the feature we selected from the list of
    // features we can use at the next tree level
    let remainingFeatures = features |> Map.remove bestName
    // ... and repeat the operation for each branch,
    // building one more level of depth in the tree
    let nextLevel =
        branches
        |> Seq.map (fun (value,group) ->
            value, growTree group label remainingFeatures)
        |> Map.ofSeq

    Stump((bestName,bestFeature),defaultValue,nextLevel)
```

到这里，大部分的工作已经完成了。现在可以从一个数据集和一组特征中学习决策树了。让我们来尝试一下！算法已经稳定，是时候将探索的步骤移到新脚本文件中了。

程序清单 6-18 第一棵“泰坦尼克”树

```
#r @".\packages\FSharp.Data.2.2.2\lib\net40\FSharp.Data.dll"
#load "Tree.fs"

open System
open FSharp.Data
open Titanic
open Titanic.Tree

type Titanic = CsvProvider<"titanic.csv">
type Passenger = Titanic.Row

let dataset = Titanic.GetSample ()

let label (p:Passenger) = p.Survived

let features = [
    "Sex", fun (p:Passenger) -> p.Sex |> Some
    "Class", fun p -> p.Pclass |> string |> Some
```

```
    "Age", fun p -> if p.Age < 7.0 then Some("Younger") else Some("Older") ]

let tree = growTree dataset.Rows label (features |> Map.ofList)

dataset.Rows
|> Seq.averageBy (fun p -> if p.Survived = decide tree p then 1. else 0.)

>
val tree : Tree<CsvProvider<...>.Row,bool> =
  Stump
  (("Sex", <fun:features@17-2>),"male",
  map
    [("female",
      Stump
        (("Class", <fun:features@18-3>),"3",
        map
          [("1",
            Stump
              (("Age", <fun:features@19-4>),"Older",
               map [("Older", Answer true); ("Younger", Answer false)]));
// snipped for brevity
val it : float = 0.8058361392
```

好消息是，运行这段代码会生成一棵树且有明显的改进——80%的预测是正确的。不过也有一些坏消息。首先，输出的树奇丑无比。而且，我们完全忽略了迄今为止一直鼓吹的论调，完全没有用交叉验证评估工作。这需要改正。

> **警告**：和第 2 章中讨论的简单贝叶斯分类器类似，决策树学习的内容直接基于训练集中找到各个项目的频度。因此，必须牢记的是，如果数据集与“真实”群体相比有明显的失真，就不能反映其真实组成，预测可能完全不正确。

6.4.3 更漂亮的树

我们所得到的决策树至多只算是勉强可以理解。下面编写一个简单的函数，以更友好的格式显示。本质上，我们将递归遍历决策树，每当遇到新的深度时就增加缩进，随之打印输出特征和分支名称。

程序清单 6-19 显示决策树

```
let rec display depth tree =
    let padding = String.replicate (2 * depth) " "
    match tree with
    | Answer(label) -> printfn " -> %A" label
    | Stump((name,_),_,branches) ->
```

```
        printfn ""
        branches
        |> Seq.iter (fun kv ->
            printf "%s ? %s : %s" padding name kv.Key
            display (depth + 1) kv.Value)
```

下面是我们之前构建的树的结果：

```
? Sex : female
   ? Class : 1
     ? Age : Older -> true
     ? Age : Younger -> false
   ? Class : 2
     ? Age : Older -> true
     ? Age : Younger -> true
   ? Class : 3
     ? Age : Older -> false
     ? Age : Younger -> true
? Sex : male
   ? Class : 1
     ? Age : Older -> false
     ? Age : Younger -> true
   ? Class : 2
     ? Age : Older -> false
     ? Age : Younger -> true
   ? Class : 3
     ? Age : Older -> false
     ? Age : Younger -> false
```

很明显，你可以用更有趣的形式显示决策树，但是我的意见是，上述结果大体上已经足以看出模型要告诉我们的细节，例如，小于 7 岁、坐三等舱的女性，其幸存概率高于同一级别客舱中年龄较大的女性。

顺便说一句，我认为这是决策树的关键吸引力之一：它们的结果很容易被人理解。你可以向数学背景有限的人展示决策树，他们仍然能够理解。我们很快就会看到，树也有自己的局限性，但是只查看模型输出就能够理解其含义的能力不可低估。

6.5　改进决策树

我们编写了一个完全通用的算法，可以用于任何数据集，并据此构建一棵决策树，可以最佳的顺序使用你所喜欢的任何特征列表。这是否意味着，我们的工作已经结束了？并非如此。虽然，大部分复杂的代码块都已经完成，但是我们的决策树生成算法可能出错。实际上，我们几乎肯定会过度拟合训练数据，学到一个预测新数据时表现不佳的脆弱模型。

6.5.1 为什么会过度拟合？

让我们用一个异常的例子来说明可能发生的问题类型，考虑在乘客 ID 这一特征上建立模型。当然，这是一个荒谬的模型：观察不在训练样本中的乘客的唯一 ID，不可能帮助我们预测其命运。然而，如果你用该特征生成一棵树，将得到如下模型：

程序清单 6-20 乘客 ID 上的过度拟合

```
let idExample = [ "ID", fun (p:Passenger) -> p.PassengerId |> string |> Some ]

let idTree = growTree dataset.Rows label (idExample |> Map.ofList)

dataset.Rows
|> Seq.averageBy (fun p -> if p.Survived = decide idTree p then 1.0 else 0.0)
|> printfn "Correct: %.3f"

idTree |> display 0

>
Correct: 1.000

? ID : 1 -> false
? ID : 10 -> true
? ID : 100 -> false
// snipped for brevity
? ID : 97 -> false
? ID : 98 -> true
? ID : 99 -> true
```

这明显在许多方面是错误的。但是要注意，在数学上没有问题，只是说从实践的角度看，结果完全没有用处。如果我们用这个模型预测新数据，新的 ID 明显不会出现在树中，结果是预测值都是默认的未知值——然而，这个模型在模型和训练集的预测上是 100%正确的。

我们已经遇到过最后这个问题，你现在很有可能会这样想，“当然——他本应该使用交叉验证。”你是对的。我们将要使用交叉验证，但是现在说明这一点是很有用的。首先，因为这是对可能出现的严重错误的一个极简说明；其次，因为过度拟合是采用决策树方法时应该留意的重要问题。

和前面考虑的其他算法不同，树是递归学习的。因此，随着算法的推进，这种方法在越来越小的样本上一次学习一个特征，而不是在完整的训练集上一次学习所有特征。

思考熵的原理时，就会发现上述方法有一点疑问。首先，信息不会消失，在最糟糕的情况下，按照一个特征分割样本不会提供任何信息。因此，在当前的实现中，我们总是将所拥有的每个特征添加到树中，即使它不能增加任何信息也是如此。其次，因为熵基于比例，只要成分完全相同，在分析小样本和大样本时都会得出相同的结论。这也有些问题：从包含 10 个观测值的样本中得出的结论，不可能和从包含十万个观测值的样本中得出的结论有相同的置信度。

6.5.2　用过滤器限制过度的自信

我们首先解决较容易的问题——避免过分自信，在不足以得出结论的样本基础上学习。简单的方法之一是在树生成算法中添加过滤器，在树向下扩展时删除弱的候选。

考虑两种我们已经识别的情况。我们目前选择的是具有最高条件熵的特征，可以通过仅保留有正面熵增益的特征改善算法。为了执行该操作，需要几部分数据：显然需要一个特征，但是还需要一个样本和标签，才能计算熵和分割之后的熵，并检查分割之后的熵是否更好（也就是确实更低）。

如果在算法中硬编码这个规则，将按照如下方式修改现有代码：

```
let rec growTree sample label features =

let features =
    features
    |> Map.filter (fun name f ->
        splitEntropy label f sample - entropy label sample < 0.)
```

上述代码是可行的，但是按照面向对象涉及（SOLID）的说法，它不容易扩展。如果我们希望有更好的灵活性，例如，仅保留熵相对前值的改善超过用户定义阈值（比如至少 5%）的特征，该怎么办？关于我们讨论过的其他问题——即从不够大的样本中得出结论，又该怎么办？

上述代码提示了一种潜在的解决方案：在 Map.filter 中，我们应用了一个“特征过滤器”——检查特征符合给定样本和标签中的一个条件的函数。我们不需要硬编码这些条件，可以简单地将一组过滤器传递给我们的算法，只保留满足所有过滤器的特征。

我们修改 Tree 模块中的 growTree 代码，并添加几个内建过滤器。

程序清单 6-21　在树算法中注入特征过滤器

```
let entropyGainFilter sample label feature =
    splitEntropy label feature sample - entropy label sample < 0.

let leafSizeFilter minSize sample label feature =
    sample
    |> Seq.map feature
    |> Seq.choose id
    |> Seq.countBy id
    |> Seq.forall (fun (_,groupSize) -> groupSize > minSize)

let rec growTree filters sample label features =

    let features =
        features
```

```
        |> Map.filter (fun name feature ->
                filters |> Seq.forall (fun filter -> filter sample label feature))

    if (Map.isEmpty features)
    // snipped for brevity
```

应用这个更新后的算法，仍然使用和以前相同的特征，但是传入[entropyGain; leafSize 10]作为过滤器，下面是我所得到的树：

```
? Sex : female
   ? Class : 1 -> true
   ? Class : 2 -> true
   ? Class : 3
     ? Age : Older -> true
     ? Age : Younger -> false
? Sex : male
   ? Class : 1 -> false
   ? Class : 2 -> false
   ? Class : 3
     ? Age : Older -> false
     ? Age : Younger -> false
```

我们的决策树比起前一个版本少了一些深度。这很好：使用过滤器去掉了一些虚假的分支，结果是，我们有了一棵更容易理解、更可靠的树。还要注意，在男性之下的所有分支都得出相同的结论。这是否指出了算法中的问题？不一定。在找出分支之间的显著不同时，熵将决定分割，而不管这种差异对我们的决策有没有实质影响。例如，51%和 99%的幸存概率差异明显，但是这种差异对结论并没有实质影响，结论保持不变。根据你的要求，可以两种不同方式改进树——为最后的分支增加可能性，或者合并对单一问题答案完全相同的桩。

6.6 从树到森林

我们通过在算法中注入一些常识，限制了最明显的过度拟合风险，但是这一解决方案相当粗糙。由于树的递归特性，过度拟合是特别常见的问题，但是，在机器学习中它也是一般性的问题。从一个训练样本中学习，就有在一个样本上过度学习的固有风险。应用到树上的过滤器和前面的章节中提到的正规化方法，都用一种策略解决了这个问题：应用某种方式的补偿（惩罚），限制算法的学习范围。

在本节中，我们将介绍这个问题的另一个思考方向，从大不相同的角度看待问题的解决方法。你可能想过，过度拟合成为问题的原因之一是我们依赖单一训练集，试图尽可能从中挤出更多的信息，从而承担了学习到的模型过于集中于某个特定数据集的风险。我们可以尝试从不同数据集中学习，避免依赖于单一模型，而不是随意限定可能的学习范围。

6.6.1　用 k-折方法进行更深入的交叉验证

我们首先重新审视模型质量的问题。我们的数据集包含 891 位乘客，这不算太多。比如，如果使用前面的 25%进行训练，剩下的 25%用于验证，可用于训练分类器的例子不到 700 个——评估模型表现的只有大约 220 个。而且，正如在太少的数据上训练模型可能会造成脆弱的结果一样，在太少的数据上评估模型常常会造成虚假的结果。因为运气，某些验证样本不典型，产生了虚假的质量指标。

限制上述问题的方法之一如下：不选择单一和随机的验证样本，而是构建多个训练/验证样本，在每个样本上重复相同的过程。例如，我们可以将样本分割成 k 个“切片”，使用一个切片进行验证，剩下的 $k-1$ 个切片用于训练，创建 k 种组合。举个例子，我们可以将数据集分成 3 个一样大的切片，留下一个作为验证，建立 3 种配对（参见图 6-7），而不是简单地将样本分成 2/3 用于训练、1/3 用于验证。

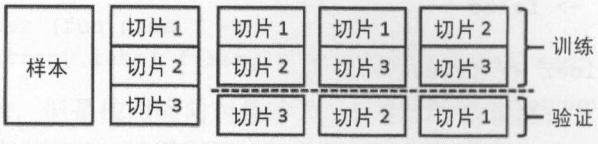


图 6-7　从一个样本中构建 3 个训练/验证组合

那么，我们为什么这么做？这样做似乎没有意义：现在，我们使用的不是一个模型和质量指标，而是有 3 个稍微不同的模型，可能有 3 个不同的质量指标，以及一个决策问题。这 3 个模型中哪一个最好？应该相信哪一个评估？这有什么益处？

如果稍微改变一下视角，就可以发现上述做法的益处。我们知道，验证指标无论如何都不是准确的：它只是个估算。在此框架下，了解估算可能出现的错误和估算本身一样重要。现在，我们不是将一切都系于一个不准确的数字上，而是了解数据中的变化如何影响模型，以及质量可能表现出来的易变性。

这种方法称作 **k-折**（**k-fold**）。k 的选择是一种妥协。在极端的情况下，如果我们将 k 增大到最大的可能值——数据集的大小，就会得到 k 个不同模型，每个只在单一例子上验证，这通常称作“留一法”。显然，k 越大，操作成本越高，因为我们的模型需要训练。而且，k 越大，每个模型共享的数据越多，因此可能表现得更像使用整个数据集的模型。

下面在我们的模型上尝试。首先，需要生成 k 对训练和验证样本。下面的实现虽然并不出色，但是确实能完成这项工作：对于 k 个值中的每一个，计算验证切片的起始和结束索引，返回 k 个样本配对的列表，每个样本都在一个元组中包含训练和验证部分。

程序清单 6-22　从一个样本中生成 k 个训练/验证配对

```
let kfold k sample =

    let size = sample |> Array.length
```

```
    let foldSize = size / k

    [ for f in 0 .. (k-1) do
        let sliceStart = f * foldSize
        let sliceEnd = f * foldSize + foldSize - 1
        let validation = sample.[sliceStart..sliceEnd]
        let training =
            [|
                for i in 0 .. (sliceStart - 1) do yield sample.[i]
                for i in (sliceEnd + 1) .. (size - 1) do yield sample.[i]
            |]
        yield training,validation
    ]
```

在基本决策树上尝试k-折分析，取 k=10：

程序清单 6-23　评估 10 个切片

```
let folds = dataset.Rows |> Seq.toArray |> kfold 10
let accuracy tree (sample:Passenger seq) =
    sample
    |> Seq.averageBy (fun p ->
        if p.Survived = decide tree p then 1.0 else 0.0)

let evaluateFolds =
    let filters = [ leafSizeFilter 10; entropyGainFilter ]
    let features = features |> Map.ofList
    [for (training,validation) in folds ->
        let tree = growTree2 filters training label features
        let accuracyTraining = accuracy tree training
        let accuracyValidation = accuracy tree validation

        printfn "Training: %.3f, Validation: %.3f" accuracyTraining accuracyValidation
        accuracyTraining, accuracyValidation]
>
Training: 0.802, Validation: 0.697
Training: 0.788, Validation: 0.843
Training: 0.796, Validation: 0.775
Training: 0.797, Validation: 0.764
Training: 0.803, Validation: 0.798
Training: 0.798, Validation: 0.843
Training: 0.804, Validation: 0.787
Training: 0.805, Validation: 0.775
Training: 0.788, Validation: 0.843
Training: 0.794, Validation: 0.787
val evaluateFolds : (float * float) list =
 [(0.8017456359, 0.6966292135); // snipped // ; (0.7942643392, 0.7865168539)]
```

这里有两个有趣的结果。首先，虽然没有巨大的不同，但是训练集上的准确率约为 79.7%，比验证集（79.1%）高出 0.6 个百分点。其次，验证样本上的质量计量指标比训练样本上的变化更大。训练集上的准确率范围为 78.8%～80.5%，而验证结果范围为 69.7%～84.3%。这并不奇怪，因为每个验证样本都更小，因而更易变。这些结果非常有用，更好地表达了我们从模型中应该得到的结果。如果我们只进行了基本的交叉验证，就只能看到 k-折方法所产生结果的 1/10。根据我们恰好选择的训练/验证配对，可能得出不同的结论——可能过分乐观，也可能过分悲观。相比之下，利用 k-折方法，我们可以得出结论，模型很可能没有过度拟合（训练集和验证集的平均误差很相近）。我们还得到了模型好坏的更准确感觉：我们的预期准确率大约为 79.1%，标准差——以该值为中心的平均误差——为 4.5 个百分点。

我希望这个简单的例子能够说明，简单地从原始数据集创建多个样本，就能够更好地反映模型的质量。这种技术主要用于评估模型的更改是否真正改进了质量。在下一节中，我们将看到如何使用类似的思路改进模型本身。

6.6.2　将脆弱的树组合成健壮的森林

利用 k-折方法，我们看到，重新采样训练集可以在不添加新观测值的情况下从中得到更多信息。我们没有将所有鸡蛋放在一个篮子里（将所有观测值放在一个训练集中），而是生成多个训练样本，使我们对模型在数据变化上的敏感性有了更好的总体认识。

我所发现的这一结果显而易见又令人着迷。说它令人着迷是因为，从表面上，我们似乎凭空创造了附加信息。利用开始时就拥有的相同信息，得以榨出全新的更好信息。说它显而易见，是因为这只是焦点的变化。从根本上讲，我们应对的是不确定性，对不确定性的最好描述就是分布。通过从原始数据中构造多个随机样本，可以模拟存在于尝试预测的现象中的易变性类型，从而复制（并计量）输出中产生的易变性。

同一个思路也可以更进一步，用于产生更健全的预测模型。我们不试图找出一个完美的模型，而是接受意外，接受输入的易变性，根据不同的训练集创建多种模型。其中一些模型是正确的，一些是错误的，但是我们将它们的预测放到一起，不太可能同时出现错误。例如，可以采取多数表决方法，希望得到“群众智慧”的效果，少数几个奇怪的预测将会被大部分合理的预测所抵消。这种通用方法称作“集成方法”，该方法研究的是将多个弱预测模型组合成强于各部分总和的整体模型。

在决策树的特例中，达到上述效果的方法之一是简单地从原始数据中统一地随机采样观测值，生成多个训练集，其中利用了置换方法，相同的观测值可能出现多次。该方法称作“Bagging”[自助聚集法（Bootstrap aggregating）的简写]。直觉上，这种方法将缓解过度拟合的风险，因为每个模型将在不同样本上训练。每个模型仍然会拾取数据中的虚假信息，但是各不相同，在组合起来时可能相互抵消。

在实现时，决策树有另一个不那么明显的问题。因为生成规程每次推进一个特征，有些特征可能会屏蔽其他特征的影响。想象一下，某个特征与另一个特征有强相关（比如，乘坐

的客舱等级和票价），自身也包含一些信息，一旦决策树选择了第一个特征，很有可能就不会选择第二个，因为剩下的信息本身不够强。结果是，我们按照熵的顺序从所有特征中选择，从而带来了一种风险：可能系统性地忽略能够传达某些有用信息的特征。

我们可以采用类似于 bagging 方法的方式随机采样特征，解决这个问题。至少在某些情况下，“隐含”特征将在没有竞争对手的情况下出现，如果它们包含较大的信息量，就有机会参与决策树。

这两种思路可以合而为一。我们不创建单一的决策树，而是创建多棵较简单的树，组成一个森林，每棵树使用随机选择的训练样本和特征的一个子集。我们依靠的不是一棵树，而是从所有树中进行多数表决。每个单独的模型都弱于一次使用所有数据的决策树，但是听起来可能有些不可思议，将它们组合起来，就能提供更可靠的预测。

6.6.3 实现缺失的部分

扩展当前代码，用森林代替树，是相当简单的。基本上，我们只需要添加 3 个要素：从一个列表中随机选择特征（没有重复），从原始样本中随机选择样本（可以重复），将多棵树组的预测组合成多数表决。我们将在 Tree.fs 文件中添加所有必要的代码，然后在“泰坦尼克”示例上尝试。

我们从第二个要素开始。假定初始样本是一个数组，有重复选择很简单，只是根据我们需要的次数选择一个随机索引：

```
let pickRepeat (rng:Random) proportion original =
    let size = original |> Array.length
    let sampleSize = proportion * float size |> int
    Array.init sampleSize (fun _ -> original.[rng.Next(size)])
```

无重复选择稍微困难一些。一种可能性是打乱整个集合，选择最先的元素。另一种可能性是递归处理集合，逐个决定应该选择的元素。如果我想要从 N 个元素当中选择 n 个，表头元素被选中的概率为 n/N。如果选中它，就要从 $N-1$ 个剩下的元素中选择 $n-1$ 个；反之，则从 $N-1$ 个中选择 n 个元素。下面是我们的代码：

```
let pickNoRepeat (rng:Random) proportion original =

    let size = original |> Seq.length
    let sampleSize = proportion * float size |> int

    let init = ([],size)
    original
    |> Seq.fold (fun (sampled,remaining) item ->
        let picked = List.length sampled
        let p = float (sampleSize - picked) / float remaining
        if (rng.NextDouble () <= p)
```

```
            then (item::sampled,remaining-1)
            else (sampled,remaining-1)) init
    |> fst
```

最后，森林的预测结果就是树最常得出的决策：

```
let predict forest observation =
    forest
    |> Seq.map (fun tree -> decide tree observation)
    |> Seq.countBy id
    |> Seq.maxBy snd
    |> fst
```

6.6.4 发展一个森林

让我们把所有代码结合起来：唯一需要的是按照用户定义数量创建树的函数。我们将采用更紧凑的方式，直接返回一个可用于预测的函数。虽然单独的树可以检查，十分实用，但是我们不打算观察几百棵树，从中找出模式和其他信息。

程序清单 6-24 从决策树创建森林

```
let growForest size sample label features =

    let rng = Random ()

    let propFeatures =
        let total = features |> Seq.length |> float
        sqrt total / total

    let featSample () = pickNoRepeat rng propFeatures features
    let popSample () = pickRepeat rng 1.0 sample
    let filters = [ leafSize 10; entropyGain ]

    let forest = [
        for _ in 1 .. size ->
            let sample = popSample ()
            let features = featSample () |> Map.ofList
            growTree filters sample label features ]

    let predictor = predict forest
    predictor
```

我们通过预先为过滤器选择合理的默认值简化了函数签名，还选择了样本占群体的默认比例（100%）和特征数量（特征总数的平方根）。提供细粒度版本不是很复杂，只需要编写扩展的函数签名，从简化的“默认”版本中调用即可。

6.6.5 尝试森林

我们已经有了算法，下面尝试一下，将结果与前面得到的树比较。我们将创建一个特征列表，使用和前面相同的 10 个训练/验证集对，训练一个由 1000 棵树组成的森林（每次使用稍有不同的特征和观测值组合），然后计算算法在训练集和验证集上的准确度。

程序清单 6-25 测试森林

```
let forestFeatures = [
    "Sex", fun (p:Passenger) -> p.Sex |> Some
    "Class", fun p -> p.Pclass |> string |> Some
    "Age", fun p -> if p.Age < 7.0 then Some("Younger") else Some("Older")
    "Port", fun p -> if p.Embarked = "" then None else Some(p.Embarked) ]

let forestResults () =

    let accuracy predictor (sample:Passenger seq) =
        sample
        |> Seq.averageBy (fun p ->
            if p.Survived = predictor p then 1.0 else 0.0)

    [for (training,validation) in folds ->

        let forest = growForest 1000 training label forestFeatures

        let accuracyTraining = accuracy forest training
        let accuracyValidation = accuracy forest validation

        printfn "Training: %.3f, Validation: %.3f" accuracyTraining accuracyValidation
        accuracyTraining,accuracyValidation ]

forestResults ()
>
Training: 0.803, Validation: 0.753
Training: 0.800, Validation: 0.775
Training: 0.808, Validation: 0.798
Training: 0.800, Validation: 0.775
Training: 0.798, Validation: 0.798
Training: 0.792, Validation: 0.854
Training: 0.800, Validation: 0.775
Training: 0.812, Validation: 0.764
Training: 0.792, Validation: 0.854
Training: 0.804, Validation: 0.820
val it : (float * float) list =
  [(0.8029925187, 0.7528089888); // snipped //; (0.8042394015, 0.8202247191)]
```

结果是，从平均水平上高于使用树的方法，但是并不特别引人注目。然而，有趣的是在训练集和验证集上计量的准确度之间的差别远比以前小。在树的例子中，准确性范围为 69.7%～84.3%；对于森林，范围窄得多，从 75.3%～85.4%。这是很好的：意味着森林通常更不容易出现过度拟合，模型可以很好地推广。也就是说，它在前所未见的新数据上的表现应该不会明显不同于在训练集上的表现。

注意：森林与普通的树相比还有一个好处，训练可以在完整的数据集上进行，而无须留出验证集。每个观测值都只用在生成的某些树中，所以人们可以使用不包含该观测值的树进行训练，并以它们为森林预测该示例的标签，检查预测是否正确。在每个示例上重复该过程，将产生所谓的“袋外估算”，这是交叉验证的一种形式。袋外估算的思路很简单，但是实现有些乏味，所以我们没有在这里包含该内容，而是将它留作众所周知的“读者练习”。

6.7　我们学到了什么？

本章涵盖了许多领域。

首先，我们介绍了决策树。决策树的吸引力部分来自于其直观性。说到底，树就是一系列问题，设计用于尽快产生诊断结论。它们模拟人们的决策方法，非常容易解读，甚至非专业人士也能理解。听起来似乎很愚蠢，但是这种方法却很有价值。在完成机器学习模型的开发之后，你往往需要让其他人相信模型的有效性。当你的模型意义可以直观阐述时，交流就会容易得多。

然而，树作为工具箱中的一员也有技术上的原因。我们作为基础的“泰坦尼克”数据集很典型——非常凌乱。它包含了不同类型的特征，从数值（票价）到分类（男性/女性），一些观测值还有漏失——但是，很容易构建一个模型处理这些潜在问题，不会带来太多的问题。树很灵活，适用于任何数据集。

除了刚刚提到的例子之外，树还可以用于 2 个以上标签的分类，这方面的应用没有任何特殊的难点。决策树方法和许多经典分类方法有一个有趣的差异，那些方法本质上都是二分法，处理 3 个或者 4 个标签时都需要更高的复杂度，就这一点而言，只要稍做努力，决策树甚至可以用于处理回归问题。

额外的好处之一是，树直接基于概率，不仅生成预测，还能够得出对应的概率，这种能力非常方便。另一方面，模型的质量对训练样本质量很敏感。如果训练样本不能合理地复制一般群体的构成，学习算法将在不正确信息的基础上做出结论，类似于我们在第 2 章中讨论的简单贝叶斯模型。

在此过程中，我们需要一致的方法，量化特征中的信息，以便决定将重点放在哪些特征上。我们介绍了香农熵——对数据集杂质含量（以及就此做出的预测可靠性）的计量，并扩展到条件熵，计量提出一个问题之后，得到更好信息的可能性。

我们还通过离散化数值型特征，阐述了熵的另一种应用方法。从构造上来讲，树依靠分类型特征运作。也就是说，区分有限离散情况的特征。因此，数值型的输入必须归纳为离散的范围，熵提供了决定用于提取最多信息的合适截止值的一种方法。

最后，我们还讨论了决策树由于递归特性而自然产生的过度拟合倾向。从构造上讲，树逐步添加特征，随着算法的推进而运作于越来越小的样本上，因此可能根据不足以做出结论的数据而包含特征，最终形成一个脆弱的模型。这引导我们探索了与过度拟合相关的两种思路：使用 k-折方法的交叉验证，以及随机森林。使用 k-折方法，我们发现创建训练和验证样本的多种组合，可以生成一系列描述模型质量的值，提供对模型可靠性的更深理解，以及受训练数据中较小变化影响的程度。

在相同思路的基础上，我们实现了随机森林的一个简单版本，这种分类方法属于被称作“集成方法”的一类模型。我们不依赖于单一决策树，而是为每棵树使用随机选择的训练数据和特征训练许多棵树，并将它们的预测组合为多数表决。这种有些荒谬的方法选择忽略某些可用数据，在许多方面近似于“群众智慧”的想法。通过创建多个较小、较不精确的模型，每个模型使用稍有不同的信息来源，我们最终创建了一个非常健壮的总体模型，因为许多独立模型不太可能同时出错，这种模型在某种程度上胜过各个单独部分的总和。

实用链接

- Kaggle “Titanic: Machine Learning from Disaster”（泰坦尼克：灾难中的机器学习）可以在这里找到：https://www.kaggle.com/c/titanic。这是依据真正的竞赛测试技能的好机会，网页上还有几个有趣的教程。
- 如果你对随机森林感兴趣，Leo Breiman（该算法的创造者）的一个网页上有很多有用的信息：https://www.stat.berkeley.edu/~breiman/RandomForests/cc_home.htm。

第 7 章

一个奇怪的游戏

用增强学习法借鉴经验

想象你是大房间中央的某个物种，眼光所至，地板上覆盖着彩色的瓷砖。你大胆地向一块蓝色瓷砖迈出一步。哟！你感觉到一阵剧痛。蓝色瓷砖不能踩？在你左侧是一块红色的瓷砖，右侧是蓝色的瓷砖，这次该试试红的了。对了！这次发生了好的事情。似乎，红色瓷砖是“好”的，蓝色瓷砖是“坏”的。默认情况下，你可能应该避开蓝色瓷砖，选择红色的。事情也可能更加复杂——重要的是瓷砖的特殊配置。对此只有一种方法——反复尝试。尝试，确认或者撤销假定，一般来说，更多地做看上去有效的工作，而较少地做看上去会失败的事。

真实世界肯定比刚才描述的彩色瓷砖世界更复杂。然而，如果你认真思考，我们的生活与此并没有不同。我们在浩瀚的宇宙中发展，遇到未知的新情况时，我们尝试各种方案。有时候，结果立竿见影。我还清楚地记得将手指插进电源插座的那一天，我马上知道，那不是一个好的决定。有些教训需要更长的时间才能吸取。例如，当你在人前举止得当时，他们通常会报以同样的态度；当你举止无礼时，通常不会有好结果。但是这一点并不总是对的，有时候需要花费时间观察行动的效果，这就使得决策不那么显而易见。

在本章中，我们将研究如何真正地为上面提到的物种建立一个“大脑”，使它能够随着经验的积累，学习到合理的行为方式。设计一个在彩色迷宫中游荡的物种本身可能和大部分开发人员所做的不同。但是，首先，这是一个有趣的问题——其次，这是一大类类似问题的简单、有代表性的版本，可以探索几个有趣的问题和广泛适用的技术。我们将：

- 编写一个非常简单的游戏，模拟一个物种发现未知世界的行为。我们将使用增强学习方法，建立一个基本策略，使该物种能从行动的结果中学习，随着经验的积累做出更好的决策。
- 分析初始方法的缺点，相应地沿着两个主要方向改进学习算法。我们将了解如何同时考虑即时回报和长期结果，克服决策中的短视行为，以及如何通过在决策过程中注入一些随机性、鼓励探索来改进学习。

7.1　构建一个简单的游戏

我们先做个热身，按照前面概述的场景，从一个无脑的物种开始构建简单游戏。我们可以建立这个世界的模型，使之即使不算智能，至少也不是全然愚蠢的。一旦掌握了所有工作原理，就可以进一步实现更复杂的版本了。

在此，重要的不是创建一个时尚的视频游戏。与此同时，拥有一个可以观察的真实游戏是很有意思的，所以我们将采用很老套的方法，复刻一个基于控制台的游戏。首先创建一个新的 F#项目，类型为控制台应用程序，命名为 SimpleGame。创建项目之后，将看到 Program.fs 文件，其内容如下：

```
// Learn more about F# at http://fsharp.net
// See the 'F# Tutorial' project for more help.

[<EntryPoint>]
let main argv =
    printfn "%A" argv
    0 // return an integer exit code
```

这是最小的控制台应用程序，已经可以按下 F5 键运行或者调试。该程序将简单地运行 main 函数（应用程序入口点），启动一个窗口，然后立即结束执行并关闭窗口。让我们试着使这个程序更有趣吧！

游戏有两个不同的方面。我们必须建立规则模型，以处理控制台上的显示，并将游戏作为程序运行。连接好这些部分之后，还需要构建物种的“大脑”。

7.1.1　游戏元素建模

我们首先建立一些模型。我们的游戏有一个从正方形瓷砖中成长起来的物种。“物种”听起来有点侮辱性，因此从现在开始我们将其称作 Hero（英雄，主角），这个名称更讨人喜欢，也更简洁。在每一轮中，Hero 可以做 3 件事：直走、左转或者右转。为了跟踪 Hero 的行踪，需要两个附加信息：位置（例如与游戏世界左上角的距离）以及方向。

这与 F#的可区分联合配合得很好。我们添加一个文件 Game.fs，将其移到项目文件列表的顶部，并启动一个模块 Game。

程序清单 7-1　游戏元素建模

```
namespace Game

module Game =

    type Dir =
```

```
    | North
    | West
    | South
    | East

type Act =
    | Left
    | Right
    | Straight

type Pos = { Top:int; Left:int }

type Hero = { Position:Pos; Direction:Dir }
```

到目前为止，一切都相当容易。下面，我们需要一个游戏世界。为了简单起见，我们的世界是没有边缘的正方形，因此，如果你一直向北，最终会超过极限，重新出现在最南方。这简化了设计，因为无须处理到达“世界尽头”的情况：不管 Hero 处于何方，他总是能够采取 3 种行动。

我们的世界很简单——每个单元要么为空，要么包含一个宝藏，要么有一个陷阱。我们可以两种方式建立其模型，现在，我们将使用映射（大约相当于一个不可变的字典）保存单元，而且只保存非空的单元。此外，还要立刻创建一个保存整个“世界状态”的记录，以及一个保存世界大小的记录。

程序清单 7-2 游戏世界建模

```
type Cell =
    | Treasure
    | Trap

type Board = Map<Pos,Cell>

type GameState = { Board:Board; Hero:Hero; Score:int }

type Size = { Width:int; Height:int }
```

7.1.2 游戏逻辑建模

有了各个组件，就可以开始用几个函数定义游戏世界的工作方式了。我们首先需要的是能够移动 Hero：给定他的位置和方向，他（或她）在棋盘上的下一步走向何方？我们不能简单地增大或者减小位置变量，因为一旦走到世界的边缘，应该重新出现在另一侧。我们将创建自定义取模运算符%%%，以所需的方式处理负数，创建一个“调整”位置并返回保持在棋盘范围之内的等价地址，一切准备就绪——现在，可以编写一个函数 moveTo，按照 4 个可能方向中的一个，将某个位置变换为下一步的位置。

程序清单 7-3　在游戏世界中移动

```
let inline (%%%) (x:int) (y:int) =
    if x >= 0 then x % y
    else y + (x % y)

let onboard (size:Size) (pos:Pos) =
    { Top = pos.Top %%% size.Height;
      Left = pos.Left %%% size.Width; }

let moveTo (size:Size) (dir:Dir) (pos:Pos) =
    match dir with
    | North -> { pos with Top = (pos.Top - 1) %%% size.Height }
    | South -> { pos with Top = (pos.Top + 1) %%% size.Height }
    | West -> { pos with Left = (pos.Left - 1) %%% size.Width }
    | East -> { pos with Left = (pos.Left + 1) %%% size.Width }
```

建模工作接近完成了。在 Hero 这一方面，我们需要的是应用决策（向左、向右或者直行）得到下一个位置，这很简单。

程序清单 7-4　实现 Hero 的移动

```
let takeDirection (act:Act) (dir:Dir) =
    match act with
    | Straight -> dir
    | Left ->
        match dir with
        | North -> East
        | East -> South
        | South -> West
        | West -> North
    | Right ->
        match dir with
        | North -> West
        | West -> South
        | South -> East
        | East -> North

let applyDecision (size:Size) (action:Act) (hero:Hero) =
    let newDirection = hero.Direction |> takeDirection action
    { Position = hero.Position |> moveTo size newDirection; Direction = newDirection }
```

最后，我们需要为游戏世界中发生的情况建模。现在，每当 Hero 到达一个位置，如果那个位置什么都没有，则不发生任何事情；如果有一个宝藏，则得到 100 分；如果是一个陷阱，则丢掉 100 分，陷阱或者宝藏随之消失。这同样很简单。函数 computeGan 搜索 Hero 在棋盘上的位置以及对应的得分，updateBorad 以当前棋盘为参数，删除 Hero 所在的单元（如果有的话）。

程序清单 7-5 更新游戏世界

```
let treasureScore = 100
let trapScore = - 100

let computeGain (board:Board) (hero:Hero) =
    let currentPosition = hero.Position
    match board.TryFind(currentPosition) with
    | Some(cell) ->
        match cell with
        | Treasure -> treasureScore
        | Trap -> trapScore
    | None -> 0

let updateBoard (board:Board) (player:Hero) =
    let currentPosition = player.Position
    board
    |> Map.filter (fun position _ -> position <> currentPosition)
```

建模已经完成，下面，我们就要运行游戏了！

7.1.3 以控制台应用的形式运行游戏

下面，我们将可以正常工作的最基本组件连接起来。游戏必须运行一个循环：要求 Hero 做出决策，相应更新世界，显示——然后重新开始。我们还需要初始化游戏世界的代码。

在 Program.fs 文件中，对默认代码做一些修改，创建一个 Program 模块，使用用于 Game 建模部分的同一个命名空间 Game：

```
namespace Game

open System
open System.Threading
open Game

module Program =

    [<EntryPoint>]
    let main argv =

        0 // return an integer exit code
```

现在，我们有了一个控制台应用程序的外壳，可以添加初始化功能了。这里没有什么特别困难的部分，我们必须决定游戏世界的大小和 Hero 的初始位置，还需要在游戏世界中放置一些陷阱和宝藏，让 Hero 的探险有些味道！我们将随机填充这个映射类型变量，每个位置有 50%的概率包含某种东西，陷阱和宝藏的概率相同。

程序清单 7-6　初始化游戏世界

```
module Program =

    // world initialization
    let size = { Width = 40; Height = 20 }
    let player = { Position = { Top = 10; Left = 20 }; Direction = North }

    let rng = Random ()

    let board =
        [ for top in 0 .. size.Height - 1 do
              for left in 0 .. size.Width - 1 do
                  if rng.NextDouble () > 0.5
                  then
                      let pos = { Top = top; Left = left }
                      let cell = if rng.NextDouble () > 0.5 then Trap else Treasure
                      yield pos, cell ]
        |> Map.ofList

    let score = 0
    let initialGameState = { Board = board; Hero = player; Score = score }

    [<EntryPoint>]
    let main argv = // etc...
```

程序缺失的另一部分是 Hero 做出决策的方法。现在，Hero 将保持“无脑”的状态：他就像无头苍蝇，在每一轮都随机决定直行、左转或者右转（本章后面将用大量时间来修改这种逻辑）。

程序清单 7-7　Hero 的初始决策

```
let initialGameState = { Board = board; Hero = player; Score = score }

// decision function
let choices = [| Straight; Left; Right |]
let decide () = choices.[rng.Next(3)]

[<EntryPoint>]
let main argv = // etc...
```

现在轮到主循环了。我们现在不担心显示问题（下一小节再考虑）——只是确保如果有事情发生，将打印出每一步的当前得分。在那种情况下，循环应该如程序清单 7-8 所示。

程序清单 7-8　在一个循环中运行游戏

```
[<EntryPoint>]
let main argv =
```

```
let rec loop (state:GameState) =

    let decision = decide ()

    // world update
    let player = state.Hero |> applyDecision size decision
    let board = updateBoard state.Board player
    let gain = computeGain state.Board player
    let score = state.Score + gain

    // world rendering
    printfn "%i" score

    let updated = { Board = board; Hero = player; Score = score }

    Thread.Sleep 20
    loop (updated)

// start the game
let _ = loop (initialGameState)

0 // return an integer exit code
```

此时，如果运行代码，应该能看到屏幕上打印的数字。Hero 做出决策并在游戏世界中走来走去，成绩随之变化——但是没有任何图形显示，这种效果完全无法引起人们的兴奋。让我们来改变这一状况，在游戏中添加一些图形。

7.1.4 游戏显示

我们需要的最后一部分是游戏的显示。这很简单——创建另一个文件 Redering.fs，将其移到 Program.fs 之上，并将所有显示代码放到一个新的模块中。我们的“图形”将依赖 System.Console，使用 Console.SetCursorPosition 确定绘图位置，使用 Console.ForegroundColor 设置绘图的颜色。

程序清单 7-9 游戏显示

```
namespace Game

open System
open Game

module Rendering =

    let offset (pos:Pos) = (pos.Left, pos.Top + 2)
    let writeAt (left,top) color (txt:string) =
```

```
        Console.ForegroundColor <- color
        Console.SetCursorPosition (left,top)
        Console.Write txt

    let prepareDisplay size =
        Console.SetWindowSize(size.Width, size.Height+2)

    let renderPlayer (before:Hero) (after:Hero) =
        writeAt (offset (before.Position)) ConsoleColor.Black "·"
        writeAt (offset (after.Position)) ConsoleColor.Yellow "·"

    let renderBoard (before:Board) (after:Board) =
        after
        |> Map.iter (fun pos item ->
            if (before |> Map.containsKey pos)
            then
                match item with
                | Treasure ->
                    writeAt (offset pos) ConsoleColor.Blue "@"
                | Trap ->
                    writeAt (offset pos) ConsoleColor.Red "+"
            else writeAt (offset pos) ConsoleColor.Black " ")

    let renderScore score =
        writeAt (0,0) ConsoleColor.White (sprintf "Score: %i " score)
```

在此要指出一点，F#有意地将我们引向一条有效的路径。我们都看到，项目从最好的意图开始，但是很快就陷入一大堆“面条式代码”中，在这种代码中，所有代码块相互依赖。大部分人都同意，虽然现实代码依赖于领域模型，但是更改显示不应该影响到领域。F#类型推理系统要求代码必须自顶向下使用（只能调用当前行之前已经编写的代码），自然地强制关注点分离，使依赖性直接可见。Rendering 模块在项目中处于 Game 模块之下，因为它依赖于后者——不可能在这两个区域之间创建不健康的代码循环。

剩下的唯一一件事情就是在游戏循环中加入显示代码。在 Program.fs 文件中加入 open Rendering 语句，在程序清单 7-8 中，用程序清单 7-10 代替 printfn "%i" score。

程序清单 7-10　在主循环中显示游戏

```
let score = state.Score + gain

// world rendering
renderScore score
renderPlayer state.Hero player
renderBoard state.Board board

let updated = { Board = board; Hero = player; Score = score }
```

现在，我们已经为游戏运行做好了准备。运行之后，应该看到类似于图 7-1 所示的画面。

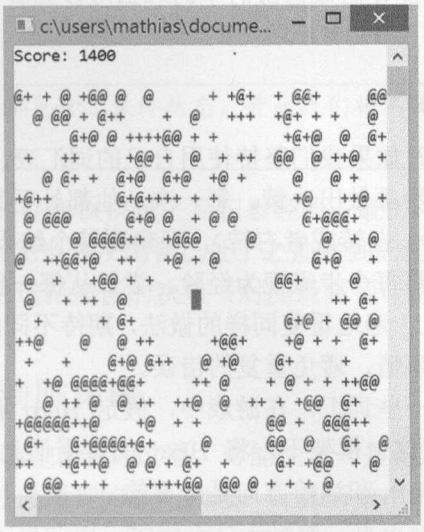


图 7-1 运行游戏

■ 注意：真实的游戏使用不同的配色方案，背景是黑色的。

我们的主角由一个小的黄色矩形表示，无意识地不断走动，随机地遇到陷阱（+）和宝藏（@）。这个游戏当然不会在游戏大会上得到任何奖项，但是现在我们有了一个基准，以及显示进展情况的手段。

7.2 构建一个粗糙的“大脑”

现在是时候给 Hero 一点智力了。开始时没有什么花哨的功能——只是从错误和成功中学习的最低智力。一旦有了这种原始的处理方法，就可以观察到过程中所学到的，并尝试使其更加智能，或者至少不那么傻。

最初，Hero 对自己成长的世界一无所知。他不知道应该避开陷阱，寻找宝藏。我们可以用巧妙的预编程例程明确地构建其“大脑”，为他提供预先确定的策略。但是，我们将采用另一条道路，这至少有两个原因。首先，编写预先确定的策略既困难又容易出错。困难是因为从一开始，设计这样的策略就没有很明显的途径。容易出错是因为，如果我们决定对游戏稍做改变，例如修改寻获宝藏的奖励，或者在游戏中添加不同类型的元素——例如另一种陷阱，就必须重写整个策略。

其次，这种方法有些浪费。每当 Hero 走动，他就会积累宝贵的经验，应该可以自行学习有效的行动和无效的行动。构建一个学习例程，观察已经采取的行动、行动的结果并逐步采取更多好的行动、减少不好的行动，比上述做法有趣得多。毕竟，这或多或少就是我们学

习的方式。此外，学习例程还有一个好处，能够解决前面提到的问题：如果我们依赖经验而不是硬编码的策略，Hero 就能在游戏修改时学习和适应。

7.2.1　决策过程建模

那么，如何做到这点呢？如果我们坚持使用自己的词汇表，这就是我们想要的：Hero 在游戏世界中来回走动，每一步都做出决策。每一次，他都处于某种状态（从世界中了解到的事物），执行某个动作（直行、左转或者右转），观察到某个结果（动作的得失），最终进入一个新的状态。合理的方法是将每个步骤视为经验，尝试从每一步行动的结果中学习。俗话说（这是争论的来源）："一遍又一遍地重复同样的做法，期待不同的结果，是疯子所为。"相反，智者一遍又一遍地做正确的事情，避免重复的错误。

我们来分解问题，确定一些词汇。在游戏中，假定 Hero 有些目光短浅，只能看到附近的情况，如图 7-2 所示。在这一框架下，将 Hero 的可用信息 State（状态）定义为其四周——以他为中心的 8 个单元——和当前方向是合理的。

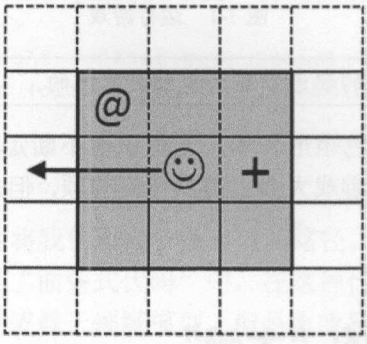


图 7-2　定义 State

我们需要的是一种学习方法，了解在给定当前所有信息（State）的情况下，Hero 应该做出什么样的决定。在这种情况下，Hero 有 3 个选择：左转、右转或者直行。我们将这些组合称作策略——当 Hero 处于特定状态下时，他有 3 种可选择的策略：

```
{当前状态; 左转}
{当前状态; 右转}
{当前状态; 直行}
```

是什么成就了一个策略的好与坏？我们的目标是让 Hero 大有收获，所以好的策略应该得到高分。那么，如何预测某个**策略**的收获？思考一下游戏的展开，可以将其视为如下形式的一系列动作：

```
状态 0 ->行动 1 -> 奖励 1 -> 状态 1 -> 行动 2 ->奖励 2 ...
```

可以重新组织上式，将游戏视为一个"经验"序列：

```
{ 状态 0; 行动 1;奖励 1; 状态 1 }
{状态 1; 行动 2; 奖励 2; 状态 3 }
...
```

■ **注意**：我们在每个经验中都包含了最终状态。开始我们将不使用该信息，但是在本章后面将会用到。

在游戏的运行中，Hero 将在每次决策之后积累经验。最初，决策有好有坏。我们希望利用这些经验学习制胜策略，因此当他发现自己处于之前遇到的情况时（已知**状态**），他可以预测 3 种可用策略的**回报**，相应选择最佳的决策（对应最高预计值的策略）。

7.2.2 从经验中学习制胜策略

现在，我们只需要评估策略的学习过程。可能性之一是记录积累的所有**经验**。当 Hero 进入一个**状态**时，可以搜索过去遇到的相同**状态**，计算所做出的每个**决策**的平均回报。这明显不现实：我们最终将积累大量数据，程序将越来越慢，最终停止。

更有效的方法是为尝试过的每个策略维护单一记录，也就是**状态**和**决策**的组合，并根据到目前为止积累的所有**经验**，逐步估算策略的好坏。当我们遇到已知**状态**时，所需要做的就是在策略目录［我们称之为**大脑**（Brain）］中查找 3 个可能匹配的**策略**，并选择具有最高值的策略。

算法几乎已经准备好了。唯一剩下的就是两个小问题的处理。第一个相当明显：Hero 在遇到未知状态时该怎么做？我为他设想的是简单地采取随机的动作，观察发生的情况。

第二个问题较复杂一些：Hero 在两次采取相同行动时应该怎么办？在特殊情况下，我们知道两次在相同条件下做相同的事将得到相同的结果——所以我们可以只采取一次行动，计量收获，并停止学习。然而，可以这么说的唯一理由是，我们知道游戏是如何实现的，不能保证一直是这种情况。例如，我们可能决定在游戏世界中宝藏和陷阱将随机出现，或者宝藏或者陷阱的回报随时改变。在那种情况下，上述方法就很不好：我们依赖单一经验（可能是不同寻常的）形成策略，并在之后完全停止学习。

更好、更稳健的一种方法（在本游戏范围之外也建议使用）是始终保持学习。模拟前一章我们所做的，有一种简单的方法：逐步改善对某个策略预期回报的估算，使用一个**学习速率** α（0～1）决定策略价值更新的激进程度。每当在一个状态下采取某种行动时，根据如下方案更新策略的价值。

$$\text{价值（策略）} \leftarrow (1-\alpha)\times \text{价值（策略）} + \alpha \times \text{收益}$$

着手实现之前，还有两点简要说明。第一，每当我介绍包含某个比率的公式时，都喜欢用如下“古怪的老把戏”解释公式的含义：将值设置为 0 和 1，看看剩下什么。在本例中，将 α 设置为 0 意味着“从不学习”，保持初始估算结果。相反，将 α 设置为 1 表示“忘记所有已经学习的”：不管之前对该策略的价值如何估算，忽略结果并用观察到的最新值（你刚刚

得到的收获）代替。两者之间的学习速率 α 在两个极端之间求得平衡：越靠近 0，调整越慢，该值越高，对新观测值的反应越快。

我们能检查算法的有效性吗？

我们可以在一个小例子上非正式地验证上述过程收敛于正确值。假定收获总为 100。使用初始估算值 0，学习速率 0.5，重复我们的更新规程，将很快地从 0 收敛到 100，该值正确：

```
> let vs = Seq.unfold (fun x -> Some(x, 0.5 * x + 0.5 * 100.)) 0.
vs |> Seq.take 20 |> Seq.toList;;
val it : float list =
  [0.0; 50.0; 75.0; 87.5; 93.75; 96.875; 98.4375; 99.21875; 99.609375;
   99.8046875; 99.90234375; 99.95117188; 99.97558594; 99.98779297; 99.99389648;
   99.99694824; 99.99847412; 99.99923706; 99.99961853; 99.99980927]
```

7.2.3　实现“大脑”

正如我们将显示代码分离到单独的模块中，清晰地将学习算法分离到单独的区域也很有意义。所以，我们在项目中添加一个包含 Brains 模块的新文件 Brains.fs，将其移到 Game.fs 之下，然后开始建模。

程序清单 7-11　“大脑”建模

```
namespace Game

open System
open Game

module Brains =

    // My current direction and surrounding cells
    type State = Dir * (Cell option) list

    type Experience = {
        State: State; // where I was
        Action: Act; // what I did
        Reward: float; // what reward I got
        NextState: State; } // where I ended

    // one possible course of action I can take
    // when I am in that State
    type Strategy = { State:State; Action:Act; }

    // Strategies I tried, and what reward
    // I should expect from them
    type Brain = Map<Strategy,float>
```

该模型几乎逐字对应我们的问题描述。我们有一个 *State*（状态）变量，组合了当前方向和一个单元（Cell）项列表，指示某些单元可能为空。*Experience*（经验）和 *Brain*（大脑）变量将每种可能的策略［*State* 和可以采取的行动（Action）］映射到一个估算的回报。

对代码进行一些重构，将 Program 中的随机决策函数（见程序清单 7-7）也移到 *Brain* 中，是很有意义的：

```
let rng = Random ()
let choices = [| Straight; Left; Right |]
let randomDecide () = choices.[rng.Next(3)]
```

下面进入学习过程。我们所想要的是加入一个新的 *Experience*，并相应地更新 *Brain*。这很容易做到：我们需要提取出 *Experience* 中采用的 *Strategy*，在我们的“大脑”中查找对应的条目，并对其应用学习速率 α。注意我们使用 brain.TryFind 的方式，该函数返回一个选项，因此可以很清晰地用简单的模式匹配（见程序清单 7-12）处理两种情况（键值存在或者不存在）。

程序清单 7-12 用新的经验更新 *Brain*

```
let alpha = 0.2 // learning rate
let learn (brain:Brain) (exp:Experience) =
    let strat = { State = exp.State; Action = exp.Action }
    match brain.TryFind strat with
    | Some(value) ->
        brain.Add (strat, (1.0-alpha) * value + alpha * exp.Reward)
    | None -> brain.Add (strat, (alpha * exp.Reward))
```

■ 注意：Map.Add 实际上是“添加或者更新”。如果键值没有找到，添加一个新条目；如果键值已经存在，则替换其原始值。

现在，我们的 Hero 已经有了大脑，可以做出决策了。我们需要的是：进入一个特定的 *State* 时，如果 *Brain* 之前没有见过这种状态，将采用随机的决策，否则，在 3 种可能的策略（从当前状态下直行、左转或者右转）中，将选择具有最大价值的策略，尚未评估的策略价值为 0。这里需要的代码较多，但是仍然相当紧凑。

程序清单 7-13 用“大脑”做出决策

```
let decide (brain:Brain) (state:State) =
    let knownStrategies =
        choices
        |> Array.map (fun alt -> { State = state; Action = alt })
        |> Array.filter (fun strat -> brain.ContainsKey strat)
    match knownStrategies.Length with
    | 0 -> randomDecide ()
    | _ ->
        choices
```

```
        |> Seq.maxBy (fun alt ->
            let strat = { State = state; Action = alt }
            match brain.TryFind strat with
            | Some(value) -> value
            | None -> 0.0)
```

我们的“大脑”几乎已经准备就绪了，最后需要一个函数根据 Hero 在棋盘上的位置提取可见的状态。下面 3 个函数的目的是提取 Hero 的方向以及周围的 8 个单元，这 8 个单元由 8 个偏移量的列表标识。例如，(−1,0) 表示相对于左上角的坐标，也就是从当前位置行向北移动一步。

程序清单 7-14　提取当前状态

```
let tileAt (board:Board) (pos:Pos) = board.TryFind(pos)

let offsets =
    [ (-1,-1)
      (-1, 0)
      (-1, 1)
      ( 0,-1)
      ( 0, 1)
      ( 1,-1)
      ( 1, 0)
      ( 1, 1) ]

let visibleState (size:Size) (board:Board) (hero:Hero) =
    let (dir,pos) = hero.Direction, hero.Position
    let visibleCells =
        offsets
        |> List.map (fun (x,y) ->
            onboard size { Top = pos.Top + x; Left = pos.Left + y }
            |> tileAt board)
    (dir,visibleCells)
```

7.2.4　测试“大脑”

这样，Hero 就有了一个大脑！接下来将其插入现有的 Program 模块。第一个变化是现在我们使用“大脑”代替随机决策——因此可以将其传递到递归状态中，以提取当前状态和做出决策开始每个循环。另一个小改动当然是，我们必须在某个时点进行某种学习。这很容易实现，只需要进行程序清单 7-15 中的更改即可。

程序清单 7-15　在游戏循环中加入学习

```
let rec loop (state:GameState,brain:Brain) =

    let currentState = visibleState size state.Board state.Hero
```

```
let decision = Brains.decide brain currentState

// world update
let player = state.Hero |> applyDecision size decision
let board = updateBoard state.Board player
let gain = computeGain state.Board player
let score = state.Score + gain

// learning
let nextState = visibleState size board player
let experience = {
    State = currentState;
    Action = decision;
    Reward = gain |> float;
    NextState = nextState; }
let brain = learn brain experience
```

最后，当我们开始游戏时，必须向它提供一个大脑，这个大脑一开始是空的：

```
let _ = loop (initial,Map.empty)
```

我们已经准备好了——运行上面的代码，应该看到和以前类似的游戏，但是随着时间的推移，Hero 犯的错误明显越来越少。那么……可以交付了吗？

少安毋躁！首先，数据在哪里？在这里，我们是科学家，要评估“大脑”是否做了正确的事情，应该更多地依靠证据，而不是“犯错明显减少”这样的模糊陈述。执行这种评估的最简方式可能是进行一次面对面的比较，看看两种方法在同等长度的游戏中的表现。因为两种方法都涉及随机性，我们希望进行多次比较。如果仅依赖于一个游戏，就可能因为运气而观察到异常的结果。多次运行游戏能够减少这种风险。

但是，这造成了另一个小麻烦。明显，运行和记录的次数越多，花费的时间越长。每帧图像的渲染至少需要 20 毫秒，500 步的游戏至少需要 10 秒。运行多次游戏所花费的时间很快就达到分钟的数量级。我们可以减少花费的时间吗？很明显，可以采用“无头”（再次致歉，我保证不再用双关语了）的方式，删除所有图形显示代码，运行无图形的游戏，简单地记录最终的得分，至少就当前的目的来说，这才是我们所关心的。

这是脚本的完美用例。我们并不真的想将这一试验产品投产，仅仅为此创建另一个控制台应用程序也有点愚蠢。因此，在项目中添加一个脚本文件，复制-粘贴 Program.fs 的全部内容——稍做修改，使其以脚本形式运行（参见程序清单 7-16）。

程序清单 7-16　模拟不同大脑的效能

```
#load "Game.fs"
open Game.Game
#load "Brains.fs"
open Game.Brains
open System
```

```
let size = { Width = 40; Height = 20 }
let player = { Position = { Top = 10; Left = 20 }; Direction = North }

let rng = Random ()

let board =
    [ for top in 0 .. size.Height - 1 do
          for left in 0 .. size.Width - 1 do
              if rng.NextDouble () > 0.5
              then
                  let pos = { Top = top; Left = left }
                  let cell = if rng.NextDouble () > 0.5 then Trap else Treasure
                  yield pos, cell ]
    |> Map.ofList

let score = 0
let initialGameState = { Board = board; Hero = player; Score = score }

let simulate (decide:Brain -> State -> Act) iters runs =

    // Loop now includes the iteration count
    let rec loop (state:GameState,brain:Brain,iter:int) =

        let currentState = visibleState size state.Board state.Hero
        let decision = decide brain currentState

        // world update
        let player = state.Hero |> applyDecision size decision
        let board = updateBoard state.Board player
        let gain = computeGain state.Board player
        let score = state.Score + gain

        // learning
        let nextState = visibleState size board player
        let experience = {
            State = currentState;
            Action = decision;
            Reward = gain |> float;
            NextState = nextState; }
        let brain = learn brain experience

        let updated = { Board = board; Hero = player; Score = score }

        if iter < iters
        then loop (updated,brain,iter+1)
        else score

    [ for run in 1 .. runs -> loop (initialGameState,Map.empty,0) ]
// Simulating different brains
```

```
printfn "Random decision"
let random = simulate (fun _ _ -> Game.Brains.randomDecide ()) 500 20
printfn "Average score: %.0f" (random |> Seq.averageBy float)

printfn "Crude brain"
let crudeBrain = simulate Game.Brains.decide 500 20
printfn "Average score: %.0f" (crudeBrain |> Seq.averageBy float)
```

对此我不做太多评论，因为它和 Program.fs 的原始代码很接近。第一个有趣的差异是我们为递归循环状态添加了迭代数量，这样每个游戏只运行有限、已知的长度。第二个差异是不在循环中硬编码决策函数，而是将其作为参数注入。该参数是一个函数，必须符合签名 decide:Brain -> State -> Act。这本质上为决策定义了一个接口——我们最终可以通过传递两种可能的决策方法，测试两种策略。

在我的机器上运行时，看到的结果如下：

```
Random decision
average: 855
Crude brain
average: 2235
```

每次运行脚本的结果都不一样，但是，通常应该观察到有“大脑”的结果好于没有“大脑”的结果，这是很好的现象。而且，如果观察每次运行的细节，就会发现结果有相当大的波动。不管有没有“大脑”，有些游戏都会演变成一场惨败。换言之，首先，我们确认“大脑”能够做一些对的事情——但是也发现，做对的事情不一定每次都能保证得到好的结果。有赢有输，但是总体来讲，长期运行之下采用后一种策略结果更好。

7.3 我们能更高效地学习吗?

我们确定简单的“大脑”聊胜于无。这是一个成功。与此同时，将 Hero 的行为描述为“智能”有点夸大其词了。我们希望从“傻得像砖”到“不完全是傻瓜”，而不是实现飞跃。例如，你可能会注意到，有时候 Hero 沿着直线不停地走，而不顾并不太远处的大量宝藏。一方面，这当然是避开陷阱的一种可靠战术；另一方面，这也肯定不是最好的做法，因为它完全牺牲了潜在的收获。

如果我们想要一个更聪明的 Hero，能够做到吗？有希望——但是为此需要更多地思考学习和知识的含义，更好地理解当前模型中的不足之处以及改进的方法。

7.3.1 探索与利用的对比

作为软件工程师，你可能常常会面对一个问题：我应该使用熟悉的技术，还是应该尝试之前从未用过的新奇思路（我觉得这些思路很有趣，但是可能会完全失败）？

简而言之，这就是探索与利用之间的矛盾：你可以选择利用已知的策略，这些策略之前已经尝试过，结果可以预测。也可以探索新的可能性，希望学到比当前更好的新策略，但是需要承担失败的风险，一旦失败，情况就会比保守的“一切照常”方法更糟。作为软件工程师，如果全部都采用前一种方法，那到现在我们还要在穿孔纸带上编码；而如果都采用后一种方法，我们就要不断地全部更换知识栈和过程。

这也是我们的 Hero 常常“自逐其尾”的原因。他所做的唯一探索是被迫的，因为他从未碰到某种情况，没有任何默认选择可用。然而，一旦找到有正面结果的策略，他就会一次又一次地重复，而不会尝试任何其他方法。

这暗示了可以用来改善“大脑”的第一个方向：赋予 Hero 一些冒险精神，使其不断尝试随机策略，甚至在有合适的策略时也这么做。最简单的方法称作 **e-学习**：每一轮中，我们以一个较小的概率 e，不管已知的策略，采用随机决策进行探索。这将有助于确保我们不会在有更好策略的情况下囿于熟悉的策略。

7.3.2　红色的门和蓝色的门是否不同？

愚者一次又一次地重复相同的错误，智者则发现类似的情况，以相同的方式应对。当我第一次尝试打开红色的厨房门时，可能需要摸索才能发现它的运作方式，在下一次看到它时，我至少应该能够重复上一次的方法。但是，当我看到通用起居室的蓝色门时，尽管情况稍有不同（门是蓝色的，房间也不一样），我还是有可能尝试和打开厨房门相同的方法。

智能行为包括发现类似状态、抽象并将学到的知识应用到更广泛的环境中。让我们变得更聪明的是，不仅发现了“红色的厨房门”，而且还认识了“门”。万幸，我们不会在每次遇到一扇新门时都从头学习。

我们的 Hero 和他目前的“大脑”饱受缺乏抽象能力之苦。具体地说，请想一想，我们的游戏是对称的。举个例子，图 7-3 中显示的两种情况是完全相同的，唯一的差别是“世界的状态”向右旋转了 90° 。

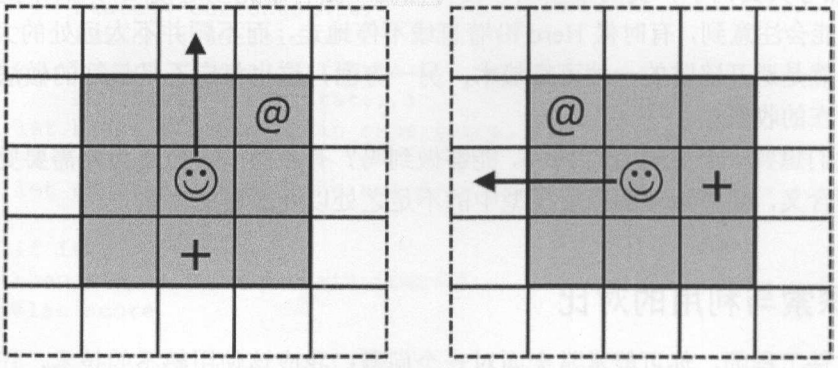


图 7-3　旋转后的状态是完全相同的

在当前的实现中，两个状态的表示截然不同，组成了大脑中的两个不同条目。如果将其与另外两种等价的状态一起归纳为单一状态，我们就在两个方面有了收获。首先，我们的“大脑”使用的内存要少得多。正如我不需要记住打开过的每一扇门的运作方式一样，Hero 只需要 25%的脑力就可以实现相同的过程。其次，他的学习也会提速很多，因为适用于一种状态的策略也可以直接用于其他 3 种情况。

我们需要做的唯一一件事是旋转和重新调整可见状态，使其面向 Hero 的视线。但是要注意，从某种程度上说，这里我们“作弊”了。初始方法的优点在于我们实际上对游戏世界的工作方式不做任何假设，简单地从观察中创建一个状态，将其视为黑盒——这种方法可行。现在，我们使用了对游戏机制的某种认识，这就不完全是黑盒法了。更难的问题（这里不做叙述）是构建一个大脑，使其自行发现等价的状态，进一步归纳相关状态。这有些离题太远，所以在此略去。

7.3.3 贪婪与规划的对比

现在，Hero 的行为可以用“贪婪”来描述——环顾四周，立即选择可以得到最高回报的决策。对他来说，这一策略还不错——但是贪婪不总是最佳的策略。现实世界的一个经典例子是这样的决定：前往高校获得更好的教育，而不是找一份不需要那种条件的工作。难点在于，贪婪的方法会导致找一份低薪的工作，立刻开始赚钱，而不是付出教育成本，后者有希望在几年以后得到更有趣、更高薪的工作机会。

两种选择都没有错，但是 Hero 从未考虑过去学校。他只对当下做出决定，毫不考虑未来——但是，事先规划、考虑短期和长期收获是所谓的“智能决策”的一部分。孤注一掷是危险的想法！

我们所喜欢的“大脑”能够平衡这两个方面。在当前状态下，我们在评估策略时只考虑了直接收益：

$$\text{价值（策略）} \leftarrow (1-\alpha)\times \text{价值（策略）} + \alpha \times \text{收益}$$

我们可以使情况更加平衡：不仅包含直接收益，还包含该策略最终结果的价值。使用的公式如下：

$$\text{价值（策略）} \leftarrow (1-\alpha)\times \text{价值（策略）} + \alpha \times (\text{收益} + \gamma \times \text{价值(下一状态的最佳选择)})$$

同样，γ 也是介于 0 和 1 之间的系数。当前模型是 $\gamma=0$ 的特例，完全忽略了下一步发生的情况。相反，将 γ 设置为 1 实际上规定直接受益和最终收益同样重要。两者之间的数字在短期收益和长期规划之间达成某种妥协。这一公式被称为 **q-学习**：它推广了我们最初的方法，这样，我们用于确定每个可能决策价值的规程就能在直接回报和长期所得之间达成妥协，从而减少了“盲目孤注一掷”的做法。

> **注意**：如果你熟悉动态编程，上述公式可能看起来很熟悉：q-学习本质上是动态编程的一种应用。类似地，如果你曾经进行过某种经济活动，可以将 γ 视为折扣率的类比——将未来的回报折现为直接回报的“利率”。

7.4　无限的瓷砖组成的世界

我们将尝试对学习过程的所有改进。但是，为了让这一切变得更有趣，我们首先对游戏本身进行一些修改。当前的游戏很简单，我们讨论过的大部分思路都无法起作用，因为游戏太短了：一旦所有陷阱或者宝藏都被消耗掉，就没有什么可学习的了。我们的 Hero 需要更有挑战性的世界。

第一个变化是，我们将不仅有陷阱和宝藏，还有随机数量的瓷砖类型，以此来增添趣味。每种瓷砖有不同的颜色和奖励，这将让我们配置较难和较简单的试验。更大的变化是，已访问单元的内容不会消失，而是整个世界都覆盖了瓷砖，当它们被访问和“消耗”时，立即由一块随机的瓷砖代替。这样，和原来逐渐清空的游戏棋盘不同，新的游戏世界将不断变化并创建新状态，游戏玩了较长时间之后仍然有可以学习的知识。完成游戏的更改之后，我们将加入刚才讨论的 3 种改进措施，并评估它们带来的差异。

我们打算实施的主要改进是不用显式可区分联合表示单元内容，而是将每块瓷砖的内容编码成表示状态的整数，并将整数映射到一个奖励数组和一个显示颜色数组。第一个需要修改的地方为 Game.fs 文件，如程序清单 7-17 所示。

程序清单 7-17　更新游戏文件

```
// code omitted
type Board = int[,]

// code omitted

let tileValues = [| -100; -50; 50; 100 |]

let computeGain (board:Board) (hero:Hero) =
    let pos = hero.Position
    let cellType = board.[pos.Left,pos.Top]
    tileValues.[cellType]

let rng = System.Random ()

let updateBoard (board:Board) (player:Hero) =
    let pos = player.Position
    let updatedBoard = board |> Array2D.copy
    updatedBoard.[pos.Left,pos.Top] <- rng.Next(tileValues.Length)
    updatedBoard
```

换言之，tileValues 数组定义了存在于游戏世界中的瓷砖数量，以及它们的价值。在这个特例中，我们将有 4 种瓷砖，涵盖了从“非常差”（-100 分）到“非常好”（100 分）的范围。我们还使用了明显更大的可能状态数量，一个单元有 4 种可能价值而非 3 种，这将

使学习变得更难。updateBoard 函数简单地创建一个新“棋盘”，用随机的新值代替 Hero 刚刚访问过的瓷砖。

Brains 模块中只需要两处小修改：State 原来是一个 Cell 选项的列表，现在则用一个整数列表代替。tileAt 函数现在简单地返回数组内容，不需要对该位置有无东西进行检查，因为每个单元都将被填充，如程序清单 7-18 所示。

程序清单 7-18 更新 Brains 文件

```
type State = int list
// code omitted
let tileAt (board:Board) (pos:Pos) = board.[pos.Left,pos.Top]
```

类似地，我们现在可以更新显示代码，进行如下修改，根据索引，用一个颜色数组表示每块瓷砖。注意，由于我们现在替换 Hero 之前所在位置的瓷砖，因此 render 函数也可以简化，因为只有两块瓷砖出现变化：Hero 的新位置和前一个位置。

程序清单 7-19 更新 Rendering 文件

```
module Rendering =

    let colors =
        [|
            ConsoleColor.DarkRed
            ConsoleColor.Red
            ConsoleColor.DarkYellow
            ConsoleColor.Yellow
        |]

    let creatureColor = ConsoleColor.White

    let offset (pos:Pos) = (pos.Left, pos.Top + 2)

    let prepareDisplay size gameState =
        Console.SetWindowSize(size.Width, size.Height+2)
        let board = gameState.Board
        for x in 0 .. (size.Width - 1) do
            for y in 0 .. (size.Height - 1) do
                let pos = { Left = x; Top = y }
                Console.SetCursorPosition (offset (pos))
                let tileType = board.[x,y]
                Console.ForegroundColor <- colors.[tileType]
                Console.Write("•")

    let render (before:GameState) (after:GameState) =
        let oldPos = before.Hero.Position
        let newPos = after.Hero.Position
        // previous player position
```

```
            Console.SetCursorPosition (offset (oldPos))
            let tileType = after.Board.[oldPos.Left,oldPos.Top]
            Console.ForegroundColor <- colors.[tileType]
            Console.Write(" • ")
            // current player position
            Console.SetCursorPosition (offset (newPos))
            Console.ForegroundColor <- creatureColor
            Console.Write(" • ")
        let renderScore score =
            Console.SetCursorPosition(0,0)
            Console.ForegroundColor <- ConsoleColor.White
            printfn "Score: %i " score
```

最后剩下的就是对 Program.fs 做一些修改。需要修改的部分很少，我们修改棋盘初始化，填充每个单元，并将 renderPlayer 和 renderBoard 合并为单一的 render 函数，如程序清单 7-20 所示。

程序清单 7-20　更新 Program 文件

```
let board = Array2D.init size.Width size.Height (fun left top ->
    rng.Next(tileValues.Length))

// omitted code
// world rendering
    let updated = { Board = board; Hero = player; Score = score }
    renderScore score
    render state updated

    Thread.Sleep 20
    loop (updated,brain)

// start the game
prepareDisplay size initialGameState
let_ = loop (initialGameState,Map.empty)
```

我们准备好了。如果此时运行游戏，应该看到与之前版本稍有不同的结果，Hero 在覆盖了彩色瓷砖的棋盘上移动。最后的变化是更新 Headless.fsx 模拟脚本，只需要对棋盘的初始化做小修改，类似于 Program.fs 中的修改，用如下语句代替原始的棋盘初始化：

```
let board = Array2D.init size.Width size.Height (fun left top -> rng.Next
(tileValues.Length))
```

说到这里，让我们进行一次模拟，看看上述修改对“粗糙大脑”的表现有何影响。我在自己的机器上进行了两次 500 步的模拟，发现采用随机决策和我们的“大脑”没有什么不同。这完全不令人吃惊，因为单元的可能状态从 3 种扩展到了 4 种。这看似是一个小变化，实际上却不是。每个单元现在可以有 4 种状态而非 3 种，可见状态包含 8 个相邻的单元，这意味着我们

从 4 个方向×3×3×…×3 个状态扩展到 4 个方向×4×4×…×4 个状态——也就是从 26244 个状态扩展到 262144 个状态。考虑到我们的 Hero 只有 500 步的机会学习该世界，他做得不好也并不意外。我们将每次模拟的步数增加，比如十万步，应该就能让他更好地学习了。显然，这需要花费长得多的时间，但是那就是获得可靠结果的代价。在我的机器上，结果如下：

```
>
Random decision
average: -703
Crude brain
average: 12480
```

7.5 实现“大脑”2.0

现在我们有了很好的基础，可以测试“大脑”的表现，并且已经得到了一个用于比较的基准，让我们尝试一下前面讨论的 3 种可能的改进手段：减少状态数量、展望未来而不只是观察短期收益，以及保留一些探索的空间。

7.5.1 简化游戏世界

考虑到我们刚刚提到的大量状态，合理的做法应该是从简化状态、删除方向、将可见状态转换为 Hero 视线方向开始。

这些改进并不复杂。有两处地方需要修改：State 的定义，它仅包含 8 个整数组成的列表（8 个周围的单元）；visibleState 函数，它提取状态。我们需要做的是旋转状态：如果 Hero 面向背面，则保持 State 不变。如果面向西方，则将所有单元顺时针旋转 90°，就像 Hero 面向北方一样——依此类推。

这只需要对坐标进行一个变换。例如，列表中的第一个坐标是（−1,−1），表示在当前位置左上角的单元（距离顶部−1，距离左侧−1）。如果 Hero 面向西方而非北方，我们就不能读取坐标为（−1,−1）的单元，而是（1,−1）的单元。类似地，Hero 右上方的单元坐标在面向北方时为（−1,1），在面向西方时应该变换为（−1,−1）。

经过一番脑力体操，我们就可以悟出根据当前方向如何重新映射坐标。程序清单 7-21 中的 rotate 函数进行这种重排，剩下的工作就是将该函数注入 visibleState 函数，这样我们就不会应用原始坐标，而是预先将其转换为反映当前 Hero 方向的坐标：

程序清单 7-21 通过旋转减少状态数量

```
let rotate dir (x,y) =
    match dir with
    | North -> (x,y)
    | South -> (-x,-y)
    | West -> (-y,x)
```

```
    | East -> (y,-x)

let visibleState (size:Size) (board:Board) (hero:Hero) =
    let (dir,pos) = hero.Direction, hero.Position
    offsets
    |> List.map (rotate dir)
    |> List.map (fun (x,y) ->
        onboard size { Top = pos.Top + x; Left = pos.Left + y }
        |> tileAt board)
```

这就完成了。我们运行“无头”模拟，看看是否有改进。在我的机器上，结果如下，改进很明显，当然，每次运行的结果都不同：

```
>
Better brain
average: 30220
```

和原来的随机决策（平均得分−703）和“粗糙大脑”（得分为 12480）相比，这种方法似乎很成功。

7.5.2　预先规划

我们来看看，是否能够使 Hero 不那么短视，从而改善结果。首先，在真正实现之前，预先考虑下一个步骤能否改善结果？在“大脑”的原始形式中，它实际上只考虑直行、左转或者右转能否产生直接收益。在那个框架下，例如 Hero 面向北方，图 7-4 中描述的两种状态将得到同样的评估，因为在这两种情况下，直行都可以得到 100 分的直接收益。

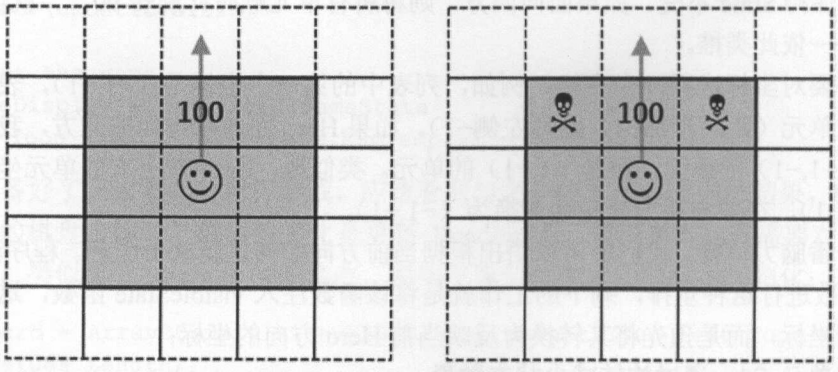


图 7-4　有完全相同的短期收益、但是最终状态大不相同的两种状态

但是，这里有一个问题。只允许直行、左转和右转 3 种移动方式，使得两种情况完全不同。在第一种情况下（左图），一旦 Hero 直行，不管发生什么，他都有两条开放路径：一条向左，一条向右。即使第三条路径（当前未知的单元）有陷阱，他仍然有许多选择。

右图描述的环境则大相径庭，得到 100 分之后，左右两侧都没有开放路径。唯一好的情况是第 3 个单元（目前无法看到）是开放路径，这无法确定——Hero 有可能完全没有好的下一步可走。

这说明，只是贪婪地关注直接收益是不充分的方法，因为这会导致我们在大不相同的两种情况下做出相同的决策，而这两种情况本应得到不同的评估。这一问题可以通过加入前面描述的 γ 项直接解决，不仅考虑一次移动得到的直接收益，还考虑下一步发生的情况。

我们实现上述更改，观察理论和实践是否一致。同样，唯一需要更改的是 Brains 模块。我们需要做两件事：创建一个函数，告诉我们给定状态下最佳移动的价值；在学习函数中包含 γ 项。这只需要增加两行代码（见程序清单 7-22）。

程序清单 7-22　为学习规程增加前瞻性

```
let alpha = 0.2 // learning rate
let gamma = 0.5 // discount rate

let nextValue (brain:Brain) (state:State) =
    choices
    |> Seq.map (fun act ->
        match brain.TryFind { State = state; Action = act } with
        | Some (value) -> value
        | None -> 0.)
    |> Seq.max

let learn (brain:Brain) (exp:Experience) =
        let strat = { State = exp.State; Action = exp.Action }
    let vNext = nextValue brain exp.NextState
    match brain.TryFind strat with
    | Some(value) ->
        let value' = (1. - alpha) * value + alpha * (exp.Reward + gamma * vNext)
        brain.Add (strat, value')
    | None -> brain.Add (strat, alpha * (exp.Reward + gamma * vNext))
```

那么，这一招灵验吗？只有一种方法能够知道——模拟。同样，每次运行的结果都不一样，但是在我的机器上，“无头”模拟得到如下结果：

```
>
Forward-looking brain
average: 38300
```

改善不小，我们从大约 30000 分跃升到了 38000 分——比贪婪策略改进了 25%。

7.5.3　ε-学习

我们还有一条道路没有探索——ε-学习。该方法的思路是时常采用随机决策，即使在已

知策略有不错的结果时也是如此，从而促进探索。这种方法有些违反直觉——本质上，它的含义是：冒一定的风险以发现潜在的更好策略是值得的，尽管这样做有时候会在已有完全有效的策略时造成不好的决策。

实现 ε-学习最简单的方法是在每次决策时都掷骰子决定。在一定比例的情况下，我们将完全忽略目前已经学习到的知识，采取随机决策——这很容易实现，同样在 Brains 模块中进行（参见程序清单 7-23）。

程序清单 7-23　在学习规程中增加探索

```
let alpha = 0.2 // learning rate
let gamma = 0.5 // discount rate
let epsilon = 0.05 // random learning

// code omitted

let decide (brain:Brain) (state:State) =
    if (rng.NextDouble () < epsilon)
    then randomDecide ()
    else
        let eval =
            choices
            |> Array.map (fun alt -> { State = state; Action = alt })
            |> Array.filter (fun strat -> brain.ContainsKey strat)
        match eval.Length with
        | 0 -> randomDecide ()
        | _ ->
            choices
            |> Seq.maxBy (fun alt ->
                let strat = { State = state; Action = alt }
                match brain.TryFind strat with
                | Some(value) -> value
                | None -> 0.)
```

再次模拟，观察这种探索是否值得。注意事项同上，在我的机器上，观察到的结果如下：

```
>
Fully featured brain
average: 46500
```

又一次得到了提升：增加一些探索，使我们在之前的成绩（使用 q-学习）之上又改进了 20%。看起来，冒险尝试新事物的学习非常值得！

表 7-1 最后总结了我们的各个“大脑”实现的平均得分，说明每一步增加的学习功能给我们带来的收益。

表 7-1 增加各种学习功能所得改进的总结

大脑版本	平均得分
随机决策	-703
基本学习	12480
减小空间	30220
q-学习（前瞻）	38300
ε-学习（探索）	46500

7.6 我们学到了什么?

在本章中，我们采用了和前几章不同的方向。我们没有试图从数据集构建一个模型，提取代表“重要事实”的特征并将其组合成表示某种现象有效性的公式，而是简单地构建一个机制，随时提供新的经验，观察所做决策及其结果，在巩固取得正面成果的行为的同时，避开造成负面结果的策略——甚至没有尝试理解根本原因。

我们得到的算法称作 q-学习，使用如下核心方程：

价值（策略）$\leftarrow(1-\alpha)\times$价值（策略）$+\alpha\times$(收益$+\gamma\times$价值(下一状态的最佳选择))

对于系统所处的每个状态，以及每个状态中可能做出的决策，我们逐步构建和改善对每一可能行动价值的评估，平衡行动带来的短期收益和采取该决策造成的下一状态的价值。我们还通过 ε-学习加入了探索的元素——尽管对所处情况已经有了可靠的策略，仍然不时采取随机决策，确保探索所有可能策略的空间。

7.6.1 符合直觉的简单模型

在 q-学习中，我觉得最有趣的方面之一是，它很容易用与我们对知识及学习的理解相对应的词语解释。q-学习的基本思路是强化，在驴子表现好的时候奖励胡萝卜，而在表现不好的时候用棍子惩罚，这样驴子就会逐渐理解你希望它做的事情和正确的方式。类似地，q-学习让智能体做出决策，并改变自己的行为去最大限度地获得奖励、避免惩罚，而不一定需要理解任务的真实含义。

除了基本思路之外，我们还讨论了更细微的方面。我们的初始算法本质上盲目地向直接回报的方向前进。这个算法表现不错，但是我们还看到了采用不那么“贪婪”的方法——不只考虑直接回报，还考虑长期的结果——能够产生相当明显的改进，这实际上模拟了规划行为。

我们还看到，在算法中包含随机元素虽然有些违反直觉，但是确实能够造成另一种很好的改进。我觉得这种思路特别有吸引力。我们的模型是为了得出合理、机械的决策，因此，引入一些混沌状态能够得到更好的结果是始料未及的。但是，当你从人类行为的角度去看的

时候，就会引发共鸣。首先，学习新事物确实需要走出熟悉的领域，尝试不一定能成功的事物。其次，每个习惯的行为既有好处，又是障碍：虽然好的习惯是正面的，但是我们形成的有些习惯实际上是负面的，如果想要进步就必须忘记过去，这就是 ε-学习所做的。另一个需要注意是的是，虽然抓住机会进行试验可能造成或好或坏的结果，但是从长期来看是值得的，至少在风险可控的情况下是如此。如果你的尝试不成功，就不应该再次犯同样的错误。如果尝试很成功，你就得到了可以重复应用的知识，在一生中都能获益。

最后要指出一点，基本思路与过度拟合有一定的关联。过度拟合的描述之一是：非常高效地应用一种方法，却在过程中造成盲点的结果——这个问题可以通过引入某些随机性克服，例如随机自助聚合。

7.6.2　自适应机制

这种方法中，我觉得有趣的另一方面是其自适应特性。前面，我们说明了如何应用学习系数 α，作为之前的知识和新知识的权重，逐步更新我们的估算，最终收敛于正确值。有趣的是，这种机制与不断变化的世界十分相符。想象一下，在此过程中，我们更改了游戏规则，完全改变了每块瓷砖的效果，为它们提供了新的随机奖励。最好的一件事是，我们的 Hero 逐步适应了新的世界：得到新经验时，他对每种状态的估算都逐步变化，行为亦是如此。

仔细思考学习速率的工作原理，其实质就是，与新经验相比，旧的记忆变得越来越不重要。这是很好的特性，因为我们得以创建一个智能体，改变自己的行为应对环境的变化，而不需要准确地知道具体的原理。相比之下，根据一个数据集构建的模型要脆弱得多：只要世界保持不变，“秘方”就有效果——如果不是如此，不对模型进行修正，它就将发生系统性的错误。

最后，你可能对这是否真的有用感到疑惑。毕竟，本章中的例子是一个很傻的游戏，可能和专业编写的代码类型不是很相近。在我看来，这对于许多种情况都是很合适的方法，基本模型的简洁性使其很容易根据手上的问题进行调整。举个例子，我最近在完全不同的环境下，使用了一个和本章所用模型非常类似的模型（更多信息参见 https://vimeo.com/113597999）。我需要了解一台机器所能处理的并发任务数，这是一个有些困难的问题，答案取决于机器本身——它有多少核心和内存等——也可能随着时间而变化，例如，机器上启动了其他进程。我没有根据机器上发生的实际情况制作一个复杂的模型，而是简单地改变被处理的任务数量，计量每种活动水平（我的状态）下的吞吐率，用某个学习速率 α 更新估算，并按照概率 ε 随机地增减并发任务数量——这正是我们在游戏中所做的，只不过我没有运行一个“傻瓜”游戏，而是最终得到了一个在没有人工干预的情况下自动伸缩，以获得最大吞吐率的服务器。

第 8 章

重回数字

算法代码的优化和伸缩

在第 1 章中，我们一起研究了数字识别器问题，并且从头开始编写了一个分类器。在最后一章中，我们将从两个不同的角度重新审视这个问题：性能和实用工具。第 1～7 章主要着眼于实现解决不同问题的算法，在此过程中探索机器学习的概念。相比之下，本章的意图更多的是用于不同情况的一系列实用技巧。我们将使用第 1 章创建的数字识别器模型作为熟悉的参考点，说明广泛适用于其他情况的技术。

在本章中，我们将：

- 分析第 1 章中编写的代码，使代码更加高效，并利用并行处理，找出调整和改进性能的途径。
- 介绍 Accord.NET，这是一个提供不同经典机器学习算法“即用型”实现的.NET 库。我们将说明如何使用其中 3 种算法（逻辑回归、支持向量机和人造神经网络），并讨论一些基本思路。
- 介绍 m-brace.NET，这个程序库的意图是从 F#脚本环境中，以分布式风格在云中的一个机器群集上进行计算。我们将论证该库可用于处理更大的数据集，同时保留探索和设计模型的能力，以及 FSI 提供的快速反馈等问题。

8.1 调整代码

性能是机器学习的重要关注点，可能更甚于日常的业务线应用程序开发。正如其他代码一样，性能通常不是首先关注的问题：第一要务应该是设计一个正确的模型，产生准确的结果。实现了这一步，如果模型太慢，才是考虑调整的时机。

按照 Donald Knuth 的说法，“程序员浪费大量的时间思考或者担心程序中不关键部分的速度，这些尝试实际上在考虑调试和维护时有很大的负面影响。在大约 97%的情况下，我们

应该忘记小的效率问题：过早优化是灾祸的根源。但是，我们不应该放过关键的 3%。”

正如上面这段话所说的，这 3%在机器学习中相当重要。主要原因可以追溯到机器学习的原始描述：得到数据越多、表现就越好的程序。有证据表明，随着用于训练的数据库变得越来越大，简单和复杂的算法倾向于拥有相似的预测能力（Peter Novig 的谈话“The Unreasonable Effectiveness of Data”就是一个例子：https://www.youtube.com/watch?v=yvDCzhbjYWs）。在这一框架下，不管你使用何种算法，都可能从更大量的数据中学习，从而使执行速度变慢。在那种情况下，低效的实现可能受到更多的影响，从而直接影响生产率，甚至使算法无法在实践中应用。

8.1.1　寻求的目标

那么，我们的目标究竟是什么？改进现有代码的性能有几种含义。最明显的是速度：我们希望在更短的时间内得到相同的答案。另一方面是内存占用：当必需保存在内存中的数据增加时，在小数据集上工作的算法可能表现得很拙劣。

检查第 1 章中编写的最邻近算法实现，下面是它的大概工作方式：

```
训练一个分类器=
    在内存中加载 5000 个例子
分类一个图像=
    对于分类器中的 5000 个例子中的每一个
        计算图像与目标的 784 个像素的距离
    找出最小的距离
```

仅根据上述结构就能帮助我们考虑预期的行为，以及有必要集中精力的地方：

- 使用更多训练数据时，预测速度将呈线性下降。
- 高效地计算距离对速度有直接影响。
- 算法是内存密集的（我们需要预先在内存中加载所有训练示例）。

在此基础上，每当我们使用交叉验证时，必须在 500 个验证示例上迭代。这不会影响分类器的性能，但是对高效评估变化的能力有意义。如果我们将训练集和验证集都扩大 10 倍，可以预期，模型验证速度将会变为原来的 1/10，因为现在要处理两个嵌套的循环。

陈述上述问题的另一种方式是，如果想要通过提供更大数据集改进算法的性能，应该预期到速度上的显著下降，以及潜在的内存问题。我们选择的算法有重大的影响：如果我们想要精度，就必须付出预测速度的代价。这是否成为问题完全取决于业务环境。如果快速预测很重要，应该选择不同的算法，这样做的初始训练代价很高，但是只是一次性的代价，后续预测将会非常快。

下一节，我们将观察一些具有上述特性的分类器。现在，我们假定可以忍受慢速的预测，先探索改进算法的手段。

第一个显而易见的方向是确保我们的代码有合理的效率。一般来说，第一步应该使用剖析程序识别代码中的潜在热点。在此情况下，我们将把焦点放在距离的计算上，大部分工作

都发生在这里，这样做可以避免任何高代价或者不必要的工作。通常来说，这包括确保使用正确的数据类型和数据结构，可能使用命令式风格，用可变变量避免招致创建及分配新数据结构的代价。

第二个方向是利用并行处理。不管如何改进距离计算，仍然需要计算 5000 个距离。好消息是，我们没有必要顺序处理它们。两个图像之间的距离完全独立于其他距离，因此理论上可以同时处理这些距离，无须等待其他计算完成，从而将速度提高到 5000 倍。类似地，对于我们使用的距离函数，如果可以并行处理每一对像素，理论上可以将计算速度提高到 784 倍。

当然，将速度提高这么多是不现实的。但是，一般的想法仍然有效：实际上，每当你在代码中看到一个映射时，就有可能采用并行计算。

8.1.2 调整距离函数

我们首先分析距离函数，以及从中挤出一些性能提升的方法。我们将从程序清单 8-1 开始，这是第 1 章中得到的脚本，利用相同的数据集，试验改进手段。

程序清单 8-1 原始的最邻近模型

```
open System
open System.IO

type Observation = { Label:string; Pixels: int[] }
type Distance = int[] * int[] -> int
type Classifier = int[] -> string

let toObservation (csvData:string) =
    let columns = csvData.Split(',')
    let label = columns.[0]
    let pixels = columns.[1..] |> Array.map int
    { Label = label; Pixels = pixels }

let reader path =
    let data = File.ReadAllLines path
    data.[1..]
    |> Array.map toObservation

let trainingPath = __SOURCE_DIRECTORY__ + @"..\..\..\Data\trainingsample.csv"
let training = reader trainingPath

let euclideanDistance (pixels1,pixels2) =
    Array.zip pixels1 pixels2
    |> Array.map (fun (x,y) -> pown (x-y) 2)
    |> Array.sum
```

```
let train (trainingset:Observation[]) (dist:Distance) =
    let classify (pixels:int[]) =
        trainingset
        |> Array.minBy (fun x -> dist (x.Pixels, pixels))
        |> fun x -> x.Label
    Classify

let validationPath = __SOURCE_DIRECTORY__ + @"..\..\..\Data\validationsample.csv"
let validation = reader validationPath

let evaluate validationSet classifier =
    validationSet
    |> Array.averageBy (fun x -> if classifier x.Pixels = x.Label then 1. else 0.)
    |> printfn "Correct: %.3f"

let euclideanModel = train training euclideanDistance
```

第一步，我们需要创建一个基准。激活定时器，并执行 5000 次距离计算。我们并不真的关心输出，也不想招致和距离计算本身无关的任何代价，因此选择两个随机图像，并添加一条 ignore()语句，避免在 FSI 窗口中造成混乱：

```
#time "on"

let img1 = training.[0].Pixels
let img2 = training.[1].Pixels

for i in 1 .. 5000 do
    let dist = euclideanDistance (img1, img2)
    ignore ()
```

在我的工作站上（具有充足内存的 4 核 i7 机器），结果如下：

```
>
Real: 00:00:00.066, CPU: 00:00:00.062, GC gen0: 22, gen1: 0, gen2: 0
val it : unit = ()
```

首先要注意，这一计量结果在多次运行中可能稍有变化。我们计量的是相当短的时间，很可能有一些噪声污染观测值。为了缓解这种风险，应该至少运行几次测试，也可以进行超过 5000 次的计算。这一结果也不完全准确；如果使用编译为 dll 的相同代码，而不是在脚本中执行，可能得到更好的结果，因为编译中可能进行了额外的优化。但是，这是一个很好的出发点。计量提供的两部分信息是时间（花费 66 毫秒完成计算，CPU 时间为 62 毫秒）和垃圾收集（简称为 GC，进行了 22 次第 0 代垃圾收集，没有更高代次的收集）。垃圾收集是有关我们所创建对象多少的有趣指标；更高代次的 GC 通常不是好事。

我们可以做的第一件事是使用 pown 检查是否招致了某种代价。让我们做个简化：

```
let d1 (pixels1,pixels2) =
    Array.zip pixels1 pixels2
    |> Array.map (fun (x,y) -> (x-y) * (x-y))
    |> Array.sum

for i in 1 .. 5000 do
    let dist = d1 (img1, img2)
    ignore ()
```

```
Real: 00:00:00.044, CPU: 00:00:00.046, GC gen0: 22, gen1: 0, gen2: 0
val it : unit = ()
```

简化的效果实际上不能忽略，这可能有些令人吃惊：我们从 CPU: 00:00:00.062 下降到 CPU: 00:00:00.046。在预测一个图像时，这可能只称得上是一个微小的优化，但是在 500 个验证图像上运行该算法时，这将从 33 秒的时间中减去大约 11 秒。

我们仍然观察到一些 GC 发生。元凶最有可能是一次 zip 和一次 map 操作，这两个操作都创建了新的数组。我们将其减少到一次操作——map2，实质上是将两个操作在一遍中完成，用一个数组代替两个数组：

```
let d2 (pixels1,pixels2) =
    (pixels1, pixels2)
    ||> Array.map2 (fun x y -> (x-y) * (x-y))
    |> Array.sum

for i in 1 .. 5000 do
    let dist = d2 (img1, img2)
    ignore ()
```

```
Real: 00:00:00.016, CPU: 00:00:00.015, GC gen0: 3, gen1: 0, gen2: 0
```

这真是一个不小的改进！仍然进行了少量垃圾收集，但是明显减少了，从之前的 GC gen0: 22 下降到 GC gen0: 3，还将计算时间从最初的版本减少了大约 75%。让我们看看，是否能够完全跳过中间数组，用递归代替，从而完全摆脱 GC。我们将维护一个累加器 acc，按照索引检查像素，将差值加到累加器中，直到最后一个像素：

```
let d3 (pixels1:int[],pixels2:int[]) =
    let dim = pixels1.Length
    let rec f acc i =
        if i = dim
        then acc
        else
            let x = pixels1.[i] - pixels2.[i]
            let acc' = acc + (x * x)
            f acc' (i + 1)
```

```
    f 0 0

for i in 1 .. 5000 do
    let dist = d3 (img1, img2)
    ignore ()

Real: 00:00:00.005, CPU: 00:00:00.000, GC gen0: 0, gen1: 0, gen2: 0
```

现在，计算时间已经减少了 92%，不需要任何垃圾收集。为这一改进付出的代价是，代码显著加长，可能更难以理解。最后尝试一下：完全采用命令式风格，直接改变变量，代替累加器的传递：

```
let d4 (pixels1:int[],pixels2:int[]) =
    let dim = pixels1.Length
    let mutable dist = 0
    for i in 0 .. (dim - 1) do
        let x = pixels1.[i] - pixels2.[i]
        dist <- dist + (x * x)
    dist

for i in 1 .. 5000 do
    let dist = d4 (img1, img2)
    ignore ()

Real: 00:00:00.004, CPU: 00:00:00.000, GC gen0: 0, gen1: 0, gen2: 0
```

这和前一次尝试实际上没有任何差别。计算时间已经到了毫秒级，所以我建议在做出结论之前运行更多的计算。如果这样做（如用 100 万次迭代代替 5000 次），就会看到性能的差别并不明显。那时，两个版本的主要区别是风格，选择某一种方法的原因是代码易理解，而非性能。

我对使用可变变量的基本观点是两面的。我倾向于在默认情况下避免使用，除非有明显的好处。可变变量会带来复杂性；代码通常更难跟踪，也更难并行化。另一方面，如果你的算法涉及大型数组（机器学习中常见的情况）等，使用映射并左右复制数组将招来严厉的惩罚。在这种情况下，我将采用可变变量，但是几乎总是以对外界隐蔽的方式进行。在我们的例子中，当 dist 可变时，所有变化都发生在函数作用域内，程序的其他任何部分都不能改变它。那样，可变变量就是一个实现的细节：对于调用我的函数的任何人来说，一切似乎都是不可变的，因为不能篡改任何共享状态。

注意：还有另一个方向可能有助于加速该算法。使用不同的数据结构（如 KD-树）代替数组保存示例，可能加速近邻的搜索。我们将忽略这种可能性，因为它的实现需要的代码太多了。但是要记住，使用适合于任务的数据结构是提高算法速度或者内存效率的另一种方法。

8.1.3 使用 Array.Parallel

现在，让我们从另一个角度——并行性来看待问题。我们将关注评估函数，该函数检查500个验证图像，分类这些图像，并将其与真实的预期标签比较。原始实现使用 Array.averageBy，将两个步骤合而为一：首先，根据预测正确与否将每个图像映射为 1.0 或者 0.0；然后，计算整个数组的平均值。映射函数不是直接可见的，而是隐含的。

很明显，每个图像的预测相互独立，所以应该可以并行执行该操作。我们希望的是对训练样本“分而治之”，例如，根据机器所拥有的核心数均分分割，单独运行每一块，并将结果重组为单一的数组。

其实，F#在 Array.Parallel 模块中提供了一些现成的有趣选择，该模块输出 Array 函数的一个子集，以并行方式实现。首先，计量初始评估所花费的时间，以及距离函数中调整之后的情况：

```
let original = evaluate validation euclideanModel
>
Correct: 0.944
Real: 00:00:31.811, CPU: 00:00:31.812, GC gen0: 11248, gen1: 6, gen2: 1

let updatedModel = train training d4
let improved = evaluate validation updatedModel
>
Correct: 0.944
Real: 00:00:02.889, CPU: 00:00:02.890, GC gen0: 13, gen1: 1, gen2: 0
```

现在，我们重写 evaluate 函数，将 Array.averageBy 分解为 Array.Parallel.map 和 Array.average：

```
let parallelEvaluate validationSet classifier =
    validationSet
    |> Array.Parallel.map (fun x -> if classifier x.Pixels = x.Label then 1. else 0.)
    |> Array.average
    |> printfn "Correct: %.3f"

>
Correct: 0.944
Real: 00:00:00.796, CPU: 00:00:03.062, GC gen0: 13, gen1: 1, gen2: 0
```

计算时间从 2.9 秒下降为 0.8 秒，还要注意 CPU 时间从 2.9 变成了 3 秒：CPU 必须进行的工作大致相同。我们的时长下降超过 2/3。考虑到机器有 4 个核心，最多可以预期有 4 倍的增长，因此这一结果相当不错，特别是，得到这样的结果几乎不需要更改代码。

那么，为什么不在所有地方都使用 Array.Parallel.map？回到对并行性的描述，就会发现有一些开销：我们现在不能简单地在一个数组上只进行一遍运算，而是需要将其分割，传递给不同线程，然后重新汇集结果。因此，性能的改善与核心数量应该呈“亚线性”关系：例如，由 10 个核心分担工作所得到的加速少于（可能远远低于）10 倍。只有在每一个分块上

完成的工作量足以抵消分拆-合并的开销时，这样做才是值得的。举个简单的例子，考虑这样的情况：我们取有 1000 个元素的数组，每个元素的值都为 10，然后每个元素加 1：

```
let test = Array.init 1000 (fun _ -> 10)

for i in 1 .. 100000 do
   test |> Array.map (fun x -> x + 1) |> ignore
>
Real: 00:00:00.098, CPU: 00:00:00.093, GC gen0: 64, gen1: 0, gen2: 0
for i in 1 .. 100000 do
    test |> Array.Parallel.map (fun x -> x + 1) |> ignore
>
Real: 00:00:00.701, CPU: 00:00:01.234, GC gen0: 88, gen1: 1, gen2: 1
```

在这种情况下，仅为了加 1 而将数组分成块是不值得的，并行版本最终慢于原始版本。一般来说，这种策略适合于不可忽略的 CPU 密集任务。例如，在这一框架下，我们可以在 classify 函数中以两种方式利用并行性：将其用于距离，并行计算像素差异，或者用于示例，并行计算分类。第二个方向更有可能起作用，因为分类需要进行很多工作，而计算两个像素之间的差异可以忽略不计。

NESSOS STREAMS

Array.Parallel 以并行风格实现了 Array 模块函数的一个子集。如果需要更多，可以了解 Streams 库（http://nessos.github.io/Streams/），该库中包含了 ParStream 模块（用于并行流）中的其他函数，并提供了其他有趣的可能性。简言之，该库区分管道中的惰性和紧迫运算，将惰性运算融合在一起，可以得到性能上的好处。

8.2　使用 Accord.NET 实现不同的分类器

在前一小节中，我们讨论了提高现有算法性能的几种方法，但是，每种算法都有基本特性，定义了程序的表现。在本节中，我们将采用完全不同的方向，研究 Accord.NET，这是一个流行（且出色）的库，包含了大量机器学习和数值分析工具。

我们有双重目标。首先，编写自己的实现往往并不困难，而且会带来一些好处。特别是，你能够完全控制代码，提供了使用该代码使应用程序更紧密集成的机会，也为全局优化提供了一些空间。但是，开发机器学习算法需要大量的试验和反复尝试，有时候，你只需要快速地检查特定模型是否好于其他模型即可。在那种情况下，从头开始实现算法工作量太大，使用现成的实现可以加快淘汰过程。在这种框架下，知道如何利用 Accord 这样的丰富程序库就很实用了。

其次，我们在前面几章中介绍了许多种技术，但是也遗漏了许多其他的经典实用技术。我们无法详细介绍每一种算法，但是将在本节提供某些算法的概要介绍，并阐述这一程序库的使用方法。

8.2.1 逻辑回归

在第 4 章中，我们实现了一个经典的回归模型，尝试从多个输入特征中预测一个数字值。该方法与最邻近模型的不同点之一是过程中有一个截然不同的训练阶段。训练集用于估算分类器函数的参数，这比原始数据集要紧凑得多。如果我想要将最近邻分类器函数发送给某人，就必须发送整个训练集。相比之下，回归模型只是一个形式为 $Y= a_0+ a_1\times X_1+\cdots+a_k\times X_k$ 的函数。我所需传输的只是 $a_0,a_1,\cdots,a_k$ 的值。类似地，生成预测也明显更快：只需要进行一次简单的计算，而无须计算 5000 个距离。我们所需付出的代价是找出 $a_0,a_1,\cdots,a_k$ 的值，这是一种代价很高的训练运算。

逻辑回归本质上是相同的模型，为分类进行了改编。它的输出是 0 和 1 之间的数值，代表观测值属于两种可能类别的概率，而不是产生可取任何实数值的数字。

对于特征值 $X= [X_1; \ldots ; X_k]$的观测点，Y 的预测值通过所谓的逻辑函数生成：

$$f(X)\frac{1}{1+e^{-(a_0+a_1+x_1+\cdots+a_kx_k)}}$$

上述公式与回归模型有可见的关系：从 X 入手，我们首先计算一个线性组合的结果，这提供了值 z，并用逻辑函数 f 对 z 应用另一次变换：

$$[x_1;\cdots;x_k]\rightarrow z=a_0+a_1x_1+\cdots a_kx_k\rightarrow Y=\frac{1}{1+e^{-z}}$$

最后一个变换是为了解决如下问题：如果我们的目标是产生一个描述观测值属于某个类别概率的数字，输出最后处于[0…1]区间。这和线性回归的情况不同，线性回归可以取正负无穷大之间的任何值。但是，如果绘制 1.0 / (1.0 + exp (−z))的图形，就可以看到图 8-1 所示的形状，该函数将任何值变换为[0…1]区间中的值，z=0 时函数取值为 0.5。当线性模型预测 z=0 时，逻辑回归表示两种可能结果出现的机会为 50/50。距离 0 越远的值表明观测值属于某个类别的置信度越高。

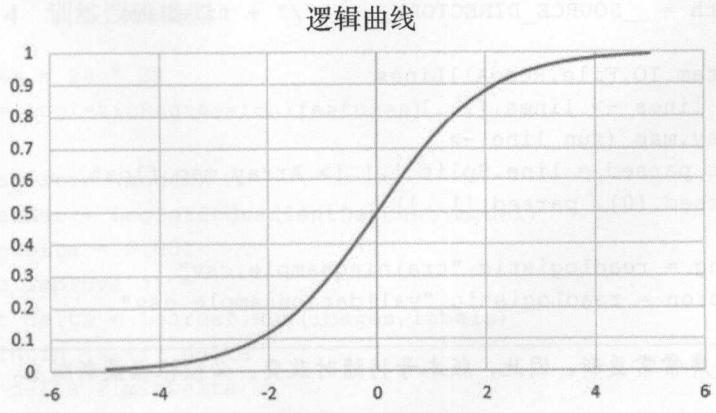


图 8-1 逻辑曲线

换言之，逻辑函数起**激活函数**的作用。它接受任何输入值，将其转换为一个二元信号：如果输入值高于 0，我们就处于状态 1，否则处于状态 0。

8.2.2　用 Accord 实现简单逻辑回归

从技术上说，估算逻辑函数参数可以用与第 4 章中类似的方式，以稍做修改的代价函数使用梯度下降方法完成。实现这种方法是有趣的练习（也不会过分复杂），但是我们将使用 Accord.NET 的版本代替。

作为出发点，我们将暂时搁置一个明显的问题：需要识别 10 个类别，但是逻辑模型只能区分两种情况。我们将首先完成一个示例，创建仅分离两种情况（比如 4 和 9）的分类器。

我们在解决方案中添加一个新的脚本文件 logistic.fsx，并添加本例将要使用的两个 NuGet 包——Accord.MachineLearning 和 Accord.Neuro。

首先，我们需要一些数据。我们将遵循和第 1 章相同的路线，但是有两处细微的不同。第一，我们将把值解析为浮点数而非整数，因为这是 Accord 所预期的；第二，我们将使用元组存储示例，标签为第一个元素，特征为第二个元素。

程序清单 8-2　读取逻辑回归所用的数据集

```
#I @"../packages"
#r @"Accord.2.15.0\lib\net45\Accord.dll"
#r @"Accord.MachineLearning.2.15.0\lib\net45\Accord.MachineLearning.dll"
#r @"Accord.Math.2.15.0\lib\net45\Accord.Math.dll"
#r @"Accord.Statistics.2.15.0\lib\net45\Accord.Statistics.dll"

open Accord.Statistics.Models.Regression
open Accord.Statistics.Models.Regression.Fitting

let readLogistic fileName =
    let path = __SOURCE_DIRECTORY__ + @"/" + fileName
    path
    |> System.IO.File.ReadAllLines
    |> fun lines -> lines.[1..]
    |> Array.map (fun line ->
        let parsed = line.Split ',' |> Array.map float
        parsed.[0], parsed.[1..])

let training = readLogistic "trainingsample.csv"
let validation = readLogistic "validationsample.csv"
```

警告：Accord 库常常更新，因此，版本号将随时改变。必须根据最新版本，相应更新脚本中的引用！

现在，内存中已经有了一个数据集，我们还需要做两件事。因为将要学习的是如何识别 4 和 9，我们将过滤数据集，只保留相关的示例。因为逻辑回归预期输出编码为 0 或者 1，我们将创建一个小的工具函数，相应地转换标签。最后，我们将用这些代码创建新的学习样本（参见程序清单 8-3）。

程序清单 8-3　为逻辑回归准备数字 4 和 9 的数据集

```
let labeler x =
    match x with
    | 4. -> 0.
    | 9. -> 1.
    | _ -> failwith "unexpected label"

let fours = training |> Array.filter (fun (label,_) -> label = 4.)
let nines = training |> Array.filter (fun (label,_) -> label = 9.)

let labels,images =
    Array.append fours nines
    |> Array.map (fun (label,image) -> labeler label,image)
    |> Array.unzip
```

现在，可以开始使用 Accord 逻辑回归模型了。Accord 主要使用面向对象风格，因此，它遵循如下模式：

1. 首先创建一个模型，定义预测的方式。
2. 将模型传递给一个学习程序——定义如何使用训练数据拟合模型的类。
3. 向学习程序提供数据，“学习”直到拟合足够好。
4. 使用原始模型实例做出预测。

在我们的例子中，将使用 LogisticRegression 模型，使用对应于扫描图像每个像素的 28×28 个特征，用 LogisticGradientDescent 学习，直到模型中的变化可以忽略。这很适合于使用递归循环（参见程序清单 8-4）。

程序清单 8-4　训练逻辑模型

```
let features = 28 * 28
let model = LogisticRegression(features)

let trainLogistic (model) =
    let learner = LogisticGradientDescent(model)
    let minDelta = 0.001
    let rec improve () =
        let delta = learner.Run(images,labels)
        printfn "%.4f" delta
        if delta > minDelta
        then improve ()
```

```
        else ignore ()
    improve ()

trainLogistic model |> ignore
```

如果运行如下代码，应该会看到 FSI 中打印输出一系列数字，表示学习过程每一步中模型参数的变化量。当变化量低于 minDelta（定义了你希望的结果与最优值之间的接近程度）时，学习停止。那时，模型就已经准备就绪。例如，可以在验证集上运行它，过滤任何不是 4 或者 9 的数字，检查哪些预测是正确的（参见程序清单 8-5）。

程序清单 8-5　验证逻辑模型

```
let accuracy () =
    validation
    |> Array.filter (fun (label,_) -> label = 4. || label = 9.)
    |> Array.map (fun (label,image) -> labeler label,image)
    |> Array.map (fun (label,image) ->
        let predicted = if model.Compute(image) > 0.5 then 1. else 0.
        let real = label
        if predicted = real then 1. else 0.)
    |> Array.average

accuracy ()
```

在我的机器上，这一算法产生了 95.4%的正确答案。注意一点：model.Compute 不一定返回 1 或者 0，而是返回一个介于 0 和 1 之间的数字，表示输出属于类 1 的概率。这就是我们检查该值高于或者低于 50%，以确定分类决策的原因。这似乎带来了额外的工作，但是实际上非常有价值：分类器不是简单地返回一个标签，还提供了预测可信度的一个估算。

8.2.3　一对一、一对多分类

如何使用二元分类器区分 10 个类别？解决这个问题有两种经典方法。第一种是“一对多”，为每个标签创建一个二元分类器，尝试辨别每个标签。在本例中，我们将创建 10 个模型：“是 0 还是其他”，“是 1 还是其他”……然后运行 10 个模型，选择置信度最高的答案，在第二种方法（一对一）中，为每个可能的标签对创建一个二元分类器（“是 0 还是 1”，“是 0 还是 2”），运行每个模型，使用循环表决进行预测，选择中选率最高的标签。

我们将实现一对多模型，主要是为了说明过程。在我们的情况下，需要 10 个不同的模型“0 或者非 0”、“1 或者非 1”，依次类推……如果我们安排数据，使每种情况下所要识别的数字标签为 1，其余为 0，则每个训练模型将返回一个数值，表示它认为该图像是某个特殊标签的强烈程度。在这个框架下，要分类一个图像，所要做的就是运行 10 个模型，选择输出值最大的模型。

这并不很复杂：在程序清单 8-6 中，对于每个类别，我们都创建了一个新的训练样本，其中的示例如果匹配所要训练的类别，则标签为 1，否则为 0。对于每个标签，我们训练一个逻辑，正如前一小节所做的那样(有一处小修改，我们将把学习过程限制在至多 1000 次迭代)，并创建一个包含所有标签及其模型的列表。整个 classifier 函数返回对应于最“可信”模型的标签。

程序清单 8-6 一对多逻辑分类器

```
let one_vs_all () =

    let features = 28 * 28
    let labels = [ 0.0 .. 9.0 ]
    let models =
        labels
        |> List.map (fun target ->
            printfn "Learning label %.0f" target
            // create training set for target label
            let trainingLabels,trainingFeatures =
                training
                |> Array.map (fun (label,features) ->
                    if label = target
                    then (1.,features)
                    else (0.,features))
                |> Array.unzip
            // train the model
            let model = LogisticRegression(features)
            let learner = LogisticGradientDescent(model)
            let minDelta = 0.001
            let max_iters = 1000
            let rec improve iter =
                if iter = max_iters
                then ignore ()
                else
                    let delta = learner.Run(trainingFeatures,trainingLabels)
                    if delta < minDelta then ignore ()
                    else improve (iter + 1)
            improve 0
            // return the label and corresponding model
            target,model)

    let classifier (image:float[]) =
        models
        |> List.maxBy (fun (label,model) -> model.Compute image)
        |> fun (label,confidence) -> label

    classifier
```

结果模型并不特别好；在我的机器上，准确率接近 83%。在此不采用其他方法，因为这段代码的差异不足以证明其价值。一方面，这两种方法都很有趣，因为它们提供了扩展二元分类器（这是你所能创建的最简单分类器）处理任意数量标签的简单技术。另一方面，必须记住，两种方法都只是启发式方法：这些方法很合理，但是不能保证有效。一对多的好处是需要的模型较少（在我们的例子中是 10 个，而一对一则要 90 个），但是两者都有潜在的问题。在一对多的情况下，我们将许多不同的情况都归入反面例子，可能造成训练的困难（例如，可以预期，区分 5 和 0 或者 6 基于不同特征）。训练一个专门区分 1 和 6 的模型比起区分 1 和 9 个不同的数字更容易。相反，在一对一的情况下，虽然训练单个模型可能更容易，但是最终决策可能很难：使用为区分两个特定数字而训练的模型，区分这两者之外的数字。（如果向区分 1 和 2 的模型传递数值 0，你认为会得出什么答案？）

8.2.4　支持向量机

我们将要研究的下一个模型是支持向量机（下称 SVM），这是另一种经典的二元分类器。它的名称很吓人，算法从概念上也比之前的复杂。我们将略去其工作细节，最大限度地减少解释。

本质上，SVM 试图用一个尽可能宽的分隔带区分两种类别，使一个类别的所有示例在分隔带的一侧，另一个类别的所有示例在另一侧。支持向量是位于（“支持”）范围两侧的示例。图 8-2 用一个简单的示例说明了这种方法的思路：两条虚线代表分隔带的边界。属于同一类（黑或者白）的所有观测值在同一侧，3 个以正方形表示的观测值是定义边界的支持向量。

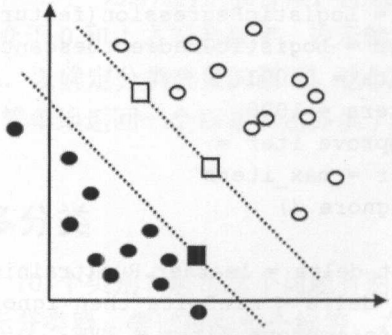


图 8-2　支持向量

正如逻辑回归，SVM 是二元分类器。与逻辑回归的一个不同之处是，标签编码为–1 和 1 而不是 0 和 1，输出就是一个分类，而不是“置信度”。下面我们来看看如何使用 Accord SVM 分类数字。Accord 有一个很方便的 SVM 多类别版本，这意味着可以直接用于 10 种分类的情况，而不需要构建额外的层次：它将为我们处理一对一的过程。

为了清晰起见，我们将把 SVM 样本放在单独的脚本 svm.fsx 中，包含用于逻辑示例的相同引用。首先，和往常一样，我们将读取数据集，但是这次不将标签转换为浮点数（见程序清单 8-7）。

程序清单 8-7 读取 SVM 所用数据

```
let svmRead fileName =
    let path = __SOURCE_DIRECTORY__ + @"/" + fileName
    path
    |> System.IO.File.ReadAllLines
    |> fun lines -> lines.[1..]
    |> Array.map (fun line ->
        let parsed = line.Split ','
        parsed.[0] |> int, parsed.[1..] |> Array.map float)

let labels,images = svmRead "trainingsample.csv" |> Array.unzip
```

训练部分需要的设置比逻辑回归方法稍微复杂一些。大部分复杂的内容在算法（algorithm）函数中。如果仔细检查代码，就会看到它以一个 SVM、一个数据集（输入和输出）以及两个值（i 和 j）为参数，返回一个训练策略。这是一次性处理多个类别所必须付出的代价：algorithm 块用于训练期间，为后台生成的每个一对一分类器创建合适的数据集和学习策略。但是，这样做的好处是可以为不同标签对配置和使用不同的策略。剩下的代码遵循类似于前面所见的逻辑：创建一个模型、一个学习程序并开始训练，这比逻辑回归更简单，因为它可以自行终止，直接返回训练样本中观察到的误差率（参见程序清单 8-8）。

程序清单 8-8 训练一个多类别 SVM 模型

```
open Accord.MachineLearning.VectorMachines
open Accord.MachineLearning.VectorMachines.Learning
open Accord.Statistics.Kernels

let features = 28 * 28
let classes = 10

let algorithm =
    fun (svm: KernelSupportVectorMachine)
        (classInputs: float[][])
        (classOutputs: int[]) (i: int) (j: int) ->
        let strategy = SequentialMinimalOptimization(svm, classInputs, classOutputs)
        strategy :> ISupportVectorMachineLearning

let kernel = Linear()
let svm = new MulticlassSupportVectorMachine(features, kernel, classes)
let learner = MulticlassSupportVectorLearning(svm, images, labels)
let config = SupportVectorMachineLearningConfigurationFunction(algorithm)
learner.Algorithm <- config

let error = learner.Run()
```

```
let validation = svmRead "validationsample.csv"

validation
|> Array.averageBy (fun (l,i) -> if svm.Compute i = l then 1. else 0.)
```

这个特殊模型在验证集上得到了92%的分类正确率，这不算太差。你想怎么样改善这一结果呢？第一个可以利用的是strategy上定义的Comlexity参数。它的值自动设置为某个水平（通常接近于1），应该能够工作得不错。增大这个数值将强制更好地拟合训练样本，但是有两个风险：可能过度拟合，或者该值过高使算法无法找出一个解，并将抛出异常。

第二个可以利用的是所谓的“核心”（Kernel）技巧。你可能注意到，在SVM设置中，我们传入一个线性（Linear）核心。其含义是，我们所搜索的是分类之间清晰、直接的分隔。如果数据集中可以找到一个这样的分隔，就没有任何问题。然而，很有可能来自每个类的示例很好地分隔，但是这种分隔并非是一条直线（或者超平面）。概略地说，“核心”技巧包括对观测值应用一个函数、将初始特征转换为更高维空间中的一组新特征，在那个空间中示例可能真正地由一个超平面分隔。如果恰好如此，我们可以训练一个“标准”线性SVM来找出这个分隔，分类新的观测值，应用同一个Kernel函数转换它们。

如果上面的解释不太清晰，不要害怕!主要的实用意义是，如果默认的线性核心无效，你可以尝试其他核心，观察是否更有效。Accord有许多内建的核心，可以探索命名空间Accord.Statistics.Kernels找到它们。

8.2.5 神经网络

在Accord中，我们要研究的最后一个分类器是人工神经网络（下称ANN）。这个名称的灵感来自大脑工作方式的大致类比。大脑是大量互相连接的神经元的集合。当我们从外部世界接收一个信号时，神经末梢发送电脉冲激活大脑中的神经元。接收足够强信号的神经元将依次发送信号给连接的神经元，最终，我们的大脑将理解信号并认识到某个事物。

人工神经网络遵循类似的结构。它们可以采用不同的形状，但是规范ANN是一组按照层次组织的神经元，每个神经元都连接到下一层次中的所有神经元。神经元本身是一个激活函数，以连接到它的每个神经元的输出作为输入，每个输入都有单独的权重，如果接收输入的加权总和足够高，将依次向前发送信号。

神经网络是一个很大的主题，仅用一个章节介绍可能是不公平的。我们将采用类似于SVM的方法，简单地说明如何用Accord的内建工具构建一个分类器，在此过程中强调几个有趣的要点。

在特殊情况下，输入信号是784个像素，所期望的输出是10个数字中（0～9）的一个。按照神经元的说法，我们可以将此表示为一个784个输入神经元的层次（每一个可以发送0～255之间的信号），最后一层有10个神经元，每个对应1个数字。我们可以构建的最简单网络是在每个输入神经元和每个输出神经元之间创建一个直接连接，如图8-3所示。

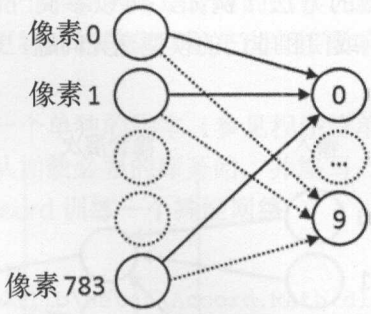


图 8-3 简单的人工神经网络

我们想要的是一个取得图像，将输入（每个像素值）发送到网络中，并将其转换为每个可能输出的 0 或者 1 值的模型。不管输入有几百个还是只有两个，都不会改变这个问题的性质，所以我们观察较简单的网络——只有两个输入和一个输出，探索如何使其工作。可能性之一是构建一个模型，如果输入信号高于某个阈值就激活一个节点，如图 8-4 所示。

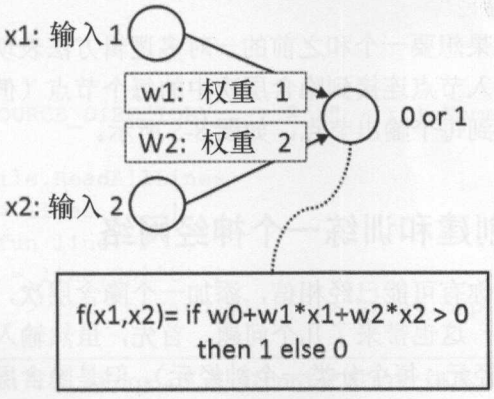


图 8-4 一个感知机

结果是一个名为“**感知机**”（Perceptron）的模型。它由 3 个元素定义：为每个输入信号分配的权重 w1 和 w2（每个输入的重要性），阈值 w0（所谓的偏置项）和一个线性激活函数 *f*，该函数组合输入，如果输入信号高于阈值则将其转换为 1（一个信号）。训练这种分类器非常简单，我们只需找出最小化误差的权重和偏置值即可——例如，使用第 4 章中讨论的类似方法。

还要注意，如果更改激活函数并在代码中使用逻辑函数代替线性组合，就复制了整个一对多逻辑模型。要点是，虽然感知机单独看起来有点简单，但是它组成了一个构件，很容易组成有趣和复杂的模型。

那么，问题是什么？人工神经网络有何与众不同之处？对 ANN 的一个关键理解是，如果简单线性模型不能很好地拟合数据，不需要使用更复杂的组件（创建新特征，使用非线性函数……），可以继续使用简单的感知机类节点，但是要在输入和输出层之间堆叠一个或者多个隐含的层次，组成更有深度的模型，如图 8-5 所示。如果你对这个问题感兴趣，有大量资源讨

论使用感知机建立逻辑门模型的方法（例如，可以参阅 http://www.cs.bham.ac.uk/~jxb/NN/l3.pdf）。事实证明，建立与门、或门和非门的模型毫无问题，但是除非添加一个隐含层次，否则异或门的模型无法建立。

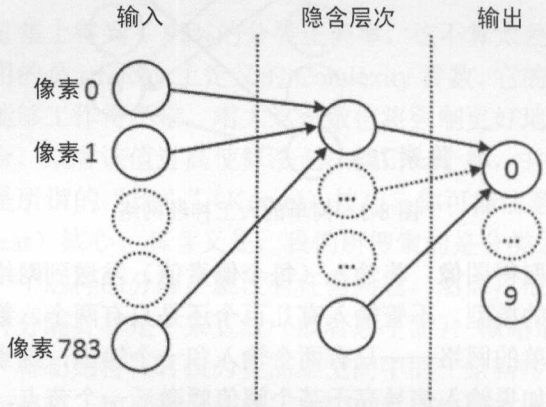


图 8-5　带有一个隐含层次的人工神经网络

在我们的例子中，如果想要一个和之前的一对多逻辑方法表现不同的模型，必须插入至少一个隐含层次。每个输入节点连接到隐含层次中的每个节点（但是没有连接到下一层次），隐含层次的每个节点连接到每个输出节点，如图 8-5 所示。

8.2.6　用 Accord 创建和训练一个神经网络

通过利用多个图示，你有可能已经相信，添加一个隐含层次，有希望将简单的感知机组合成强大的分类器。但是，这也带来了几个问题。首先，虽然输入和输出层次的大小已经明确设置（每个特征一个神经元，每个分类一个神经元），但是隐含层次应该有多少个神经元？为什么使用一个隐含层次，而不是两个、三个或者更多？

第二个有些相关的问题是，如何训练这样的模型？我们必须估算很多参数。例如，如果考虑将 784 个输入连接到 10 个神经元的层次，就必须调整将近 8000 个权重。这实在太多了。我们无法单独训练每个神经元，因为它们是相互连接的。如果更改一个神经元的输入权重，其输出就将变化，可能需要调整同一层次的其他神经元。最后，还有另一个微妙的问题：如何初始化权重？如果对每个连接从权重 0 开始，因为每个神经元是完全相同的，就有在每个神经元上学习到相同参数的危险。每个神经元没有捕捉到输入集中的不同特征，而是产生相同的输出，你可能得到重复多次的单一模型。

详细解释人工神经网络的训练本身就是一个主题，我们只限于概要的介绍。这方面的主要方法称作“反向传播”：以类似于梯度下降的方式，为网络提供示例，并比较理想值和模型输出，然后回到网络中，逐层调整每个神经元以降低误差。为了避免节点完全相同的问题，用某种随机规程初始化各个节点的权重。

在实践中，这意味着训练一个神经网络是费时的工作。虽然扩展网络深度和每个层次的节点数量，在理论上能够形成更精细的模型，但是也直接影响了训练所需的时间，更不要说过度拟合的风险了。

理解了这一点，我们再用一个单独的脚本（参见程序清单 8-9），以 Accord 创建和训练这种网络。和往常一样，我们从加载必要的库开始，并编写一个读函数以准备数据集。

程序清单 8-9 准备用 Accord 训练一个神经网络

```
#I @"../packages"
#r @"Accord.Math.2.15.0\lib\net45\Accord.Math.dll"
#r @"Accord.Neuro.2.15.0\lib\net45\Accord.Neuro.dll"
#r @"Accord.Statistics.2.15.0\lib\net45\Accord.Statistics.dll"
#r @"AForge.2.2.5\lib\AForge.dll"
#r @"AForge.Neuro.2.2.5\lib\AForge.Neuro.dll"

open Accord.Statistics
open Accord.Neuro
open Accord.Neuro.Learning
open AForge.Neuro

let nnRead fileName =
    let path = __SOURCE_DIRECTORY__ + @"/" + fileName
    path
    |> System.IO.File.ReadAllLines
    |> fun lines -> lines.[1..]
    |> Array.map (fun line ->
        let parsed = line.Split ','
        parsed.[0] |> int, parsed.[1..] |> Array.map float)
```

■ 注意：在我们的脚本中引用了 AForge，这是 Accord 使用的一组库。在本书编写的时候，Accord 正在将两个库融合为一个，这可能会使上述步骤变成多余的。

然后，我们可以设置神经网络，初始化训练阶段并验证结果（参见程序清单 8-10）。这种方法或多或少地遵循和之前看到的相同的模式，但是有几处差异值得指出。

程序清单 8-10 创建、训练和评估网络

```
let trainNetwork (epochs:int) =

    let features = 28 * 28
    let labels,images = nnRead "trainingsample.csv" |> Array.unzip
    let learningLabels = Tools.Expand(labels,-1.0,1.0)

    let network = ActivationNetwork(BipolarSigmoidFunction(), features, [| 100; 10 |])
    NguyenWidrow(network).Randomize()
```

```
        let teacher = new ParallelResilientBackpropagationLearning(network)

        let rec learn iter =
            let error = teacher.RunEpoch(images, learningLabels)
            printfn "%.3f / %i" error iter
            if error < 0.01 then ignore ()
            elif iter > epochs then ignore ()
            else learn (iter + 1)

        learn 0

        network

    let ann = trainNetwork (50)

    let validate = nnRead "validationsample.csv"
    validate
    |> Array.averageBy (fun (label,image) ->
        let predicted =
            ann.Compute image
            |> Array.mapi (fun i x -> i,x)
            |> Array.maxBy snd
            |> fst
        if label = predicted then 1.0 else 0.0)
```

我们首先需要转换输出，使每个数字成为单独特征。为了实现这一点，使用内建工具Accord.Statistics.Tools.Expand，它能将一组标签扩展为新的表现形式，每个标签成为一个单独列，本例中的两种输出被编码为-1或者1。我们使用这种特殊编码是因为所选择的激活函数BipolarSigmoidFunction：该函数的值从-1到1，其中1表示阳性（“这是一个‘1’”），-1表示阴性（“这不是‘1’”）。另外，AForge.Neuro还包含了几个内建的激活函数。

我们创建一个网络，传入激活函数、预期数量的输入（在我们的例子中是28×28个像素）以及包含在一个数组中的每层节点数，该数组的最后一个元素是预期标签数量（10）。还要注意NguyenWidrow的用法，它为网络权重创建了随机但合理的初始值。

在我的机器上，这些代码产生了将近82%的分类正确率（由于初始权重的随机性，结果可能不同）。这不算很糟，也不算很好。你可能注意到，它的运行相当缓慢。话虽如此，我们在此将训练限制在50次迭代，在第50次时误差（每一轮之后打印输出）仍然在有规律地下降。将搜索次数扩展到500次时，可以看到误差从1775下降到1200，正确率为82.6%。换言之，学习过程的速度下降了不少，但是仍然能够带来改善。

对神经网络和Accord一般用法的主题，我们的介绍到此为止。该库包含的内容远远不只在此描述的，这些有趣的示例只是说明了它的某些能力。

8.3 用 m-brace.net 实现伸缩性

按照你的愿望调整代码，到某个时点，机器将成为瓶颈。绕了一大圈，原因又回到了我们对机器学习的初始定义：好的机器学习算法在得到更多数据时应该机械地表现得更好。

上述说法的含义之一是，如果数据越多越好，那么你就会想要更多，这也就能部分地解释最近对“大数据”概念的疯狂追逐。随着数据集的增长，用简单 F#脚本进行的数据探索和模型训练将变得更慢甚至无法运行，例如，如果数据集太大，以至于无法打开或者保存到本地时。在本节中，我们将简单介绍 m-brace.net，该库通过将代码从脚本环境发送到云中的一个群集上执行，解决了上述问题。

8.3.1 用 Brisk 启动 Azure 上的 MBrace

如果我们的限制因素是计算机本身，改进方向之一是投入更多计算机。正如“9 个女人也无法在一个月中生下一个孩子”（Fred Brooks 的名言），根据问题的特性，多台计算机并不总能加快解题速度。话虽如此，一般来说，当我们在代码中识别一个映射时，应该有机会应用分而治之的方法：将映射的集合分解为块，在不同计算机上单独映射每一块，将所有块归纳为一个最终结果。

这和我们之前在 Array.Parallel.map 中看到的模式相同。我们下面将要讨论的框架 MBrace（www.m-brace.net）更进一步地扩展了这个思路：不是简单地在本地计算机的不同核心上处理分块，而是允许你通过 Microsoft Azure 云上的群集完成相同的工作。

这种方法的好处在于，所有运算仍然可以从脚本环境中进行。本质上，MBrace 的目标是让你在 Visual Studio 中使用 F#脚本，以保持快速的设计反馈循环，但是无缝地将这种代码发送到群集中远程执行。只要需要大的计算能力或者数据，就可以按需创建一个群集，而不用受到本地设备的限制，并在完成工作之后立刻关闭。

开始使用 Mbrace 的最简单方式如下：

1. 订阅 Azure（如果还没有这么做的话）。可以在 http://azure.microsoft.com 上得到免费的试用版本。

2. 在 www.BriskEngine.com 上注册一个免费账户。Brisk 服务能够帮助你轻松地在 Azure 上配给一个群集，完整地配置运行 MBrace 所需的环境，并在完成工作之后立即删除。Brisk 本身是免费的，只需为 Azure 云的使用付费。

3. 从 GitHub 上的 MBrace 存储库上下载 MBrace Starter Kit：https://github.com/mbraceproject/MBrace.StarterKit。该项目包含一个预先配置的解决方案，以及说明 MBrace 使用方法的许多示例。你可以简单地下载它，添加自己的脚本，轻松地启动项目。

此时，你应该在本地计算机上有一个如图 8-6 所示的 F#项目。我们首先在 Azure 上部署一个机器群集。为此，前往 BriskEngine.com，登录并选择 Dashboard 菜单，在那里可以检查

最近创建的群集和创建新群集。我们创建一个新群集，选择代表 MBrace 的{m}以及部署的地理位置。你可能希望部署的位置靠近自己所在地区，使机器之间的数据移动尽可能快。如果你打算访问预先存在的数据，其位置也很重要。最后，我们选择群集的大小。你可以用每小时低于 0.5 美元的价格获得 4 核的微型群集（两台中等水平的机器），也可以每小时花费 20 美元得到 256 核的最高级群集。

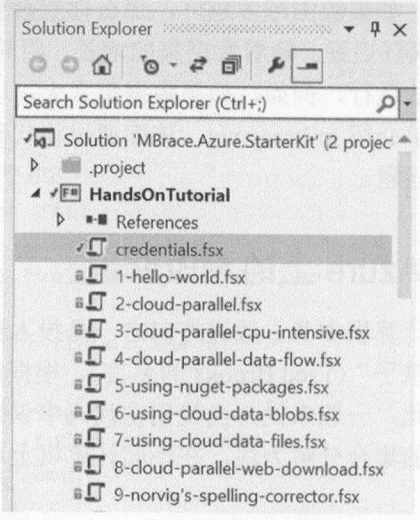


图 8-6　m-brace 启动项目

我们选择 16 核群集，这能够提供可靠的能力，每小时花费大约 1.5 美元。BriskEngine 仪表盘将指示部署所需的时间，5～10 分钟之后，你的群集就应该做好准备了（参见图 8-7）。

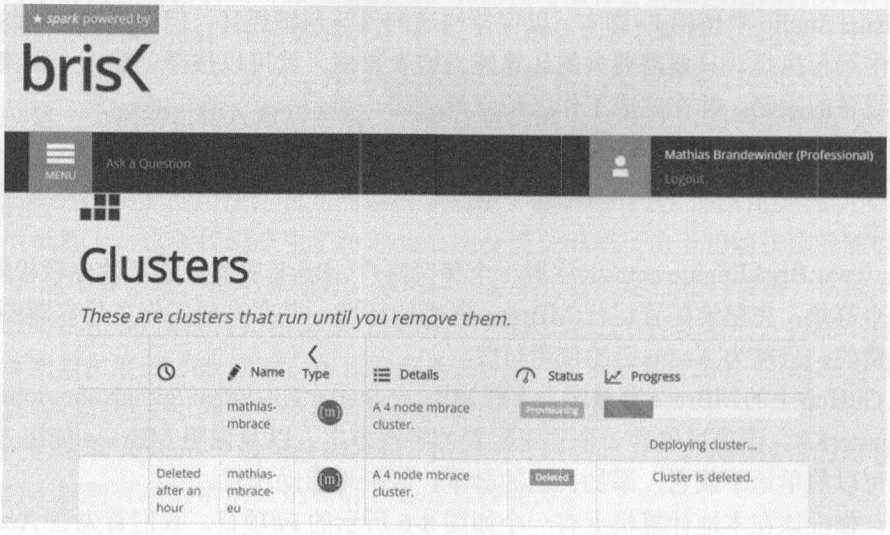


图 8-7　通过 BriskEngine 在 Azure 上部署一个群集

现在，我们已经做好在群集上运行代码的准备了，只需要从 Visual Studio 的脚本会话中连接到群集即可。我们将在 credentials.fsx 文件中为 MBrace 提供两个连接字符串，BriskEngine 使这一工作变得很简单：在仪表盘中，一旦群集激活，可以单击群集名称，将两个连接串复制到证书文件中。

最后，在 Starter Kit 解决方案中添加一个新脚本，内容见程序清单 8-11。

程序清单 8-11　连接到群集

```
#load "credentials.fsx"

open MBrace.Core
open MBrace.Azure.Client

let cluster = Runtime.GetHandle(config)
```

将这段代码发送到 FSI，然后……就可以了，你已经连接到群集了。可以运行如下代码行检查，将显示群集中运行的工作进程的相关信息：

```
cluster.ShowWorkers ()

>

 >

 Workers
 Id                    Status  % CPU / Cores   % Memory /
 --                    ------  -------------   --------------------
 MBraceWorkerRole_IN_2 Running     2.3 / 4         13.5 / 7167.0
 MBraceWorkerRole_IN_1 Running     1.4 / 4         14.1 / 7167.0
 MBraceWorkerRole_IN_0 Running     1.8 / 4         13.6 / 7167.0
 MBraceWorkerRole_IN_3 Running     1.5 / 4         13.6 / 7167.0
val it : unit = ()
```

那么，如何将工作发送给群集？通过使用云计算表达式 cloud{...}即可实现。这一名称听起来有些可怕：别吓唬自己。云计算是 F#中的一个很强大的机制，通常设计用于提供编写和管理复杂代码工作流的更简单手段。它们使用一个构建器，定义一个由{}标识的“上下文”，在这个上下文中用 let!或者 return 等关键词，隐藏一些操作的无趣副作用，强调总体工作流结构，以扩展 F#。因此，虽然理解编写新的计算表达式可能劳心劳力，但是使用现有的表达式通常并不复杂，因为要点就是让你编写更简单的代码以实现目标。

在 MBrace 中，于群集中执行代码包括两个阶段：在 cloud{...}计算表达式中定义要在云中运行的代码，然后将其发送执行。举个例子，试着逐行向 FSI 发送如下代码：

```
#time "on"
cluster.AttachClientLogger(MBrace.Azure.ConsoleLogger())
```

```
let localHello = "Hello"
let cloudHello = cloud { return "Hello" }
cloudHello |> cluster.Run
```

localHello 立即求值，但是 cloudHello 本身不做任何事情：它的类型是 Cloud<string>，是在云中运行并返回一个字符串的一段代码。它被发送给 cluster.Run 时，创建一个进程并排队等待执行。MBrace 检查已经发送到 FSI 的代码，通过网络线路发送在群集上运行代码所需的类型和数据，并在 FSI 中返回结果，就像常规的本地脚本一样。

除了发送代码执行并等待结果之外，你还可以启动一个进程等待其结果，如：

```
let work = cloudHello |> cluster.CreateProcess
cluster.ShowProcesses ()
>
Processes

Name                Process Id      Status     Completed     Execution Time
----                ----------      ------  ---------   --------------
    c93fa4d3b1e04a0093cbb7287c73feea       Completed   True      00:00:00.8280729
    c833d511623745e2b375f729be25742b   Completed   True      00:00:00.1129141

// ask if the work is complete yet
work.Completed
>
val it : bool = true
// when complete, ask for the result
Let result = work.AwaitResult ()
> val result : string = "Hello"
```

这样可以将工作发送给群集执行，并在结果可用时读取，无须阻塞 FSI。

8.3.2　用 MBrace 处理大数据集

在开始的例子中，与本地版本的“你好，世界”相比，我们从 MBrace 中得到的只是性能的下降。这并不奇怪：我们运行和本地机器相同的代码，没有任何预期收益，过程中却必须序列化所有数据。正如对于小任务来说并行化的收益无法弥补协调成本，不值得使用 Array.Parallel 一样，使用 MBrace 在小规模的问题上通常没有好处，因为在机器之间移动数据和代码带来了开销。

幸运的是，数字识别程序在这里成为了完美的例子。Kaggle 竞赛中使用的“真实”数据集包含了 50000 个例子。在目前为止的示例中使用的训练集被压缩到了 5000 个例子，这正是因为我们希望从脚本中探索思路，获得它们是否值得研究的快速反馈。如果想要保持类似的工作流，但是使用完整的数据集，该怎么办？我们首先概述运行于精简数据集之上的纯本地

代码。从程序清单 8-1 中的初始模型开始，我采用优化过的距离函数，训练一个模型并评估，如程序清单 8-12 所示。

程序清单 8-12 在精简数据集上评估模型

```
let optimizedDistance (pixels1:int[],pixels2:int[]) =
    let dim = pixels1.Length
    let mutable dist = 0
    for i in 0 .. (dim - 1) do
        let x = pixels1.[i] - pixels2.[i]
        dist <- dist + (x * x)
    dist

let optimizedModel = train training optimizedDistance

#time "on"
evaluate validation optimizedModel
```

现在，想象我们要使用所有 50000 个例子。这样做时，数据集大小只增加 10 倍，但是评估速度将要慢大约 100 倍，因为所需的运算次数从 1000×4000 增加到了 10000×40000。

这种情况相当典型：数据集变得更大通常是一件好事，但是本地计算机无法处理，开发节奏突然变慢直至停滞。这正是 MBrace 擅长的场景：我们来看看如何调整原始代码，在云中处理繁重的工作，而不需要牺牲脚本编写体验。

我们没有理由改变模型本身。第一处需要修改的代码是数据准备部分。现在，我们从单个大文件中读取，必须将数据拆分为训练集和验证集。评估函数哪里需要修改的地方最多。程序清单 8-13 描述了一种方法。我们先从代码入手，之后再做注解。

程序清单 8-13 在云中分布评估操作

```
#load "credentials.fsx"

open MBrace.Core
open MBrace.Azure
open MBrace.Azure.Client
open MBrace.Store
open MBrace.Flow

let cluster = Runtime.GetHandle(config)
cluster.AttachClientLogger(ConsoleLogger())

let fullDataPath = __SOURCE_DIRECTORY__ + @"/large.csv"

let large =
    CloudFile.Upload(fullDataPath,"data/large.csv")
```

```
    |> cluster.RunLocally

let cloudValidation =
    cloud {
        let! data = CloudFile.ReadAllLines(large.Path)
        let training = data.[1..40000] |> Array.map toObservation
        let validation = data.[40001..] |> Array.map toObservation
        let model = train training optimizedDistance
        let! correct =
            validation
            |> CloudFlow.OfArray
            |> CloudFlow.withDegreeOfParallelism 16
            |> CloudFlow.averageBy (fun ex ->
                if model ex.Pixels = ex.Label then 1.0 else 0.0)
        return correct }
```

■ **注意：在本书编写时，MBrace 仍然在发展。因此，在不久的将来 API 可能会变化。**

代码可以稍做简化，但是目前的版本突显了几个有趣的特征。首先，我们创建一个 CloudFile 文件。我们将整个数据集上传到群集，可以用 CloudFile.ReadAllLines，以类似于之前的 File.ReadAllLines 的方式读取其内容。注意 cluster.runLocally 的用法，其目的是让本地机器（原始数据文件所在位置）执行上传，而不是将其作为群集上的工作进程的任务。

cloudValidation 函数也类似于原来的纯本地版本。主要的修改是首先将评估封装在 cloud{...}中。结果是，整个过程都将在群集中运行。然后，我们不使用 Array.averageBy 评估验证版本上的模型，而是使用 CloudFlow，它的表现本质上和一个序列类似，但是将工作智能地分布到群集中。

那么，这个程序的表现如何呢？在 16 核的群集上，评估部分运行了大约 1 分钟。当然，我们必须付出初始数据上传的代价，但是这是一次性的。此时，我们可以开始修改算法测试其影响。修改后代码的每次评估将花费 1 分钟，而不是在没有使用 MBrace 时在我的计算机上测得的 5 分多钟。这一改进非常明显：在一天之中运行 15 次评估，就可以节约 1 小时，更不要说保持专注的能力了——每次尝试时，不再需要分心 5 分钟了。

MBrace 的能力远不止这个简单示例所展示的。然而，我希望它能够让你感觉到 MBrace 的实用场景——本质上，在这些示例中，你希望保持交互式、灵活的工作流，但是需要显著增加所能处理的工作量。我要专门指出，虽然这个例子是完全从脚本环境中驱动的，但是完全可以从其他来源获得数据。如果需要处理很大的数据集，这一点特别重要。在那种情况下，在脚本环境中打开数据可能不实际，甚至不可能。作为替代，可以从群集中访问任何位置的数据，在那里完成繁重的工作，并将分析结果发回本地环境。

8.4 我们学到了什么?

本章与之前我们所完成的工作有相当大的不同。我们的焦点没有放在某个特殊问题上，也没有从头开始构建一个机器学习算法以解决问题，而是重新回顾了第 1 章中开发的模型，将其作为参考点探索了两个方向：提高代码性能，阐述两种有用的建模技术。

我们从算法结构的分析开始，理解可能进行改进的地方，以及增加训练和验证所用数据时的预期性能。在这一特例中，我们确定负责计算图像间距离的函数是可以改进的瓶颈，隔离该函数，并通过计量其速度和垃圾收集特性，调整代码改善这两个部分，达到 10 倍的加速。为了这一改进，我们付出的是编写更多（也可能更复杂）代码，使用递归或者可变变量。这种妥协并不少见。虽然以函数式风格编码，偏重不可变数据结构和更高级别的函数有自身的好处（描述意图清晰及代码安全），但是也可能有缺点，最显著的就是性能。F#的优势之一是混合特性：默认情况下，这种语言倾向于函数式风格，但是也能很好地处理命令式编码，这有利于在必要时进行简单的本地优化。

我们还探索了并行性，它是榨取额外性能的一种手段。函数式风格的好处之一是有助于找出可能进行此类改进的位置。具体地说，每当使用映射时，我们就有了加速的候选，因为映射预示着操作可以并行运行。我们首先用 Array.Parallel.map 说明了这一点，之后又介绍了 m-brace.net，两者都使用上述模式，在本地机器的各个核心或者群集中的许多机器之间，对各个资源上的工作“分而治之”。

这在机器学习环境中极其实用。机器学习就是构建预测模型，在得到数据越多时表现越好。而增加模型所用的数据量是改善其表现的重要组成部分，这也造成了一个问题，因为更大的数据集需要更强大的计算能力。但是在许多情况下，数据量的增加可以得到缓解，因为数据在代码中通过一个映射或者类似的可并行操作解读，这预示着增加的容量可以分割到更多的资源上。我们看到了一个交叉验证的最好例证：即使需要评估更多的观测值，它们都可以相互独立处理，因此，可以简单地配给一个大型群集、并行化映射操作，无须花费太多精力即可处理更大的数据量。

本章的第二个主要方向是 Accord.NET 简介，这是一个丰富的.NET 机器学习库。编写自己的算法实现是完全正当的事，而且往往比看上去更容易。但是，有时候这并不现实。简单地尝试特定算法是否有效，并在无效时快速淘汰，有助于避免浪费宝贵的时间，更快地将精力集中在有前途的方向上。Accord.NET 提供了大量精心实现的分类和回归模型，是掌握这方面工作的有用工作。而且，虽然 R 类型提供程序提供了尝试现有算法的另一条道路，但是 Accord.NET 作为一个.NET 库具有优势，这使得它与.NET 代码库的集成更简单。

最后，我们对前几章中没有涵盖的 Accord.NET 中的几种经典分类器及其用法进行了介绍，这些分类器包括逻辑回归、支持向量机和人工神经网络。Accord 包含了许多其他工具，我建议深入研究这个库。我们对该库的介绍（以及作为例子的 3 种分类器）有点肤浅：在这

里详细介绍上述技术是不切实际的，但是我们希望以用法为焦点，说明 Accord.NET 遵循的一般模式，将有助于更进一步挖掘这个程序库！

实用链接

- Accord.NET 是一个出色的程序库，包含许多与机器学习算法和计算机视觉及数值分析相关的工具：http://accord-framework.net。
- MBrace，一个可伸缩云数据的脚本库：www.m-brace.net。
- Brisk，一个简化 MBrace 群集部署的网站：www.briskengine.com。
- Peter Norvig 的谈话“The Unreasonable Effectiveness of Data”：https://www.youtube.com/watch?v=yvDCzhbjYWs。
- 用单层感知机建立逻辑门模型：www.cs.bham.ac.uk/~jxb/NN/l3.pdf。

第 9 章

结语

你已经完成了本书的学习——祝贺你！这是一个漫长的旅程，我希望这也是一种享受——你可以从中学到一两个思路。在分别之前，回顾一下我们在一起实现的成就可能是值得的，我们还可能发现，有一些更广泛的主题贯穿各个章节，尽管它们之间有着很大的差别。

9.1 描绘我们的旅程

本书的目标是以容易理解和有趣的方式，向.NET 开发人员介绍机器学习的主题。机器学习是一个很大的主题，它理所当然地吸引了越来越多的注意力。这个主题常常被以抽象的方式介绍，导致很多人认为，这是一个复杂的话题，最好留给数学家去研究。

虽然数学在机器学习中有重要的作用，但是我希望，在阅读本书之后，你能够认识到，机器学习并不像看上去那么复杂，许多基本思路实际上相当简单，适用于广泛的实际问题。

让我们退后一步，看看本书介绍的基础知识。我们以概略的方式建立了机器学习问题的“地图”，澄清了重要的区别特性。首先，我们介绍了有监督和无监督方法。每种方法解决的是截然不同的问题：

- 当你不知道可能要提出的问题时，**无监督方法**帮助你了解数据。这是第 5 章的主要话题，在这一章中我们采用了 StackOverflow 问题数据集，简单地搜索模式，帮助我们理解这些难以解读的无结构数据。
- 相反，我们花费精力最多的**有监督方法**（第 1、2、4、6 章）训练一个模型，根据有标签示例，提出对我们很重要且定义良好的问题。也就是说，我们使用的是正确答案已知的数据。

在探索中，我们介绍了各种不同模型，它们之间有重要的差别。首先，我们区分了分类和回归模型，它们的差别在于所期望的答案类型。**回归**模型的目标是预测连续数值。在第 4 章中，我们开发了这样一种模型，根据不同输入预测自行车共享服务使用量水平。相反，**分类**模型确定有限数量的可能结果中最有可能的一种。我们看到了这种模型的 3 个例子，从自动识别 10 个可能数字的图像（第 1 章）到分辨非垃圾短信和垃圾短信（第 2 章），以及泰坦尼克号上旅客生存状况的预测（第 6 章）。

我们还在第7章中探索了使用增强学习的不同方法。结果模型是决定一组有限的可能行动的分类器，与之前的模型有关键的差异：我们不用过去的数据进行一次性的学习，而是构建一个模型，随着新观测值的输入而不断学习，这种方法通常称作**在线学习**。

本书自始至终都在挖掘各种真实数据集，让数据指导我们探索。尽管主题多种多样（图像、文本、数值等），但是都浮现出模式。在大部分情况下，我们最终都应用**特征提取**——将原始数据转换为许多行信息量更大或者更方便处理的数据。而且，正如我们寻求的答案类型决定了分类或者回归哪一种更合适那样，我们根据特征是连续还是分类，采用不同的方法。我们还了解了如何通过“装箱”将连续特征转换为离散特征（例如“泰坦尼克”示例中的年龄），或者反过来将分类转换为一系列指示变量（回归示例中的周日）。

9.2 科学!

我们从各个章节中看到的另一个模式涉及方法论。我们从想要回答的问题开始，收集可用的任何数据，开始进行一系列试验，逐步创建和改进与事实相符的模型。

在这个意义上，开发机器学习模型与开发常规的业务线应用程序有很大的不同。在构建应用程序时，你通常要实现一系列功能：每个功能有某种描述完成状态的验收标准。开发人员将问题分解为小的代码块，将它们组合起来，一旦所有代码块的工作符合预期，任务就完成了。

相比之下，开发机器学习程序更接近于遵循科学方法进行的研究活动。你事先并不知道特定的思路是否可行，必须描述一种理论，用所拥有的数据构建一个预测模型，然后验证构建的模型是否适合于数据。这有一些难度，因为开发这样的模型所需的时间难以估计。你尝试一个非常简单的思路，可能在半天里就完成开发，也可能花费数周之后，除了失败的试验之外一无所得。

当然，一切都不是这么明确的。开发常规应用程序也涉及某种不确定性，也会有失败的试验。但是，事实仍然是，在机器学习中，在模型与数据比较之前，你无法知道自己的想法是否可行。话虽如此，某些软件工程思路仍然适用，但是应用方法略做修改。

就像为试图提供的功能编写清晰的规格说明很有帮助一样，尽早思考计量成功的方法，然后为此目的做好准备，是至关重要的。正确的代码只是建立好的机器学习模型的一小部分，有价值的模型应该是实用的，这也意味着它必须擅长做出预测。在这个框架下，我们在本书中反复使用**交叉验证**。留出部分数据，并用于训练，一旦模型就绪，在验证集上测试其工作状态，模拟实际条件下模型得到新输入时发生的情况。在某些方面，交叉验证的作用类似于常规代码所用的测试套件，可以检查模型工作是否符合意图。

对我来说，不管是机器学习还是软件开发，尽快构建可工作原型的习惯都是很有用的。在机器学习环境中，这意味着创建所能想到的最简单、可快速执行的模型。这种做法有许多好处：迫使你组合一个从数据到验证的端到端过程，可以随时重用和改进；有助于及早捕捉潜在问题；建立一个基准——设置评判其他模型是好是坏门槛的数字。最后，如果幸运的话，这个简单模型可能工作得很好，在这种情况下你就提早完成了任务。

说到简单的模型，我希望在本书中可以说明一个要点。以机器学习为中心的大部分论述都强调花哨的模型和技术。复杂的算法很有趣，但是最终，花时间理解数据、提取合适的特征更为重要。为复杂的算法提供质量低下的数据，也不能魔法般地产生好的答案。相反，正如我们在前几章中所见到的，很简单的模型使用精心制作的特征，也能产生令人吃惊的好结果。而且，作为附加的好处，模型越简单，也就越容易理解。

9.3 F#：函数式风格更有效率

我们一起编写的绝大部分代码是以 F#编写的，这是一种函数式优先的.NET 语言。如果这是你第一次使用 F#，我希望你能够喜欢它，也希望能激励你进一步探索！在某种程度上，函数式编程苦于和机器学习类似的问题，也就是说，它常常被作为理论性和抽象的主题描述。F#成为我在过去几年中首选的语言，和理论毫无关系。我发现它是一种效率超群的语言，可以用清晰而简单的代码表达想法，并快速地改进这些想法、更快地完成任务。而且，我发现使用这种语言很有乐趣。

按照我的看法，在应用到机器学习主题时，F#表现出了很高的素质。首先，内建的脚本环境和具有轻量、表现力强的语法的语言十分关键。开发机器学习模型涉及许多探索和试验，可以一次加载数据，然后持续探索，而无须被重新加载和编译打断思路，是成功的关键。

其次，回顾我们一起构建的模型，你可能已经注意到一个通用模式。从数据源开始，我们读取和提取特征，应用学习规程更新模型直到拟合足够好，然后用交叉验证计算某些质量指标。这种通用过程和函数式风格十分契合，我们在不同问题上的实现看上去很相似：应用一个映射将数据转换为特征，使用递归应用模型更新和学习，并使用平均或者分折将预测归纳为精确度等质量指标。函数式语言的词汇表很自然地与机器学习所解决的问题类型契合。而且，还有一个附加的好处，函数式模式强调不可变数据，很容易并行化，这在处理大量数据时很方便。

上述特点也适用于其他函数式语言，但 F#还有两个特别有趣的特性。第一个是类型提供程序，我们在第 3 章中探索了这个机制。大部分语言要么采用动态类型，要么采用静态类型，每种都有自己的优势和难题。要么外部数据容易访问但是从编译程序获得的帮助有限，要么是相反的情况。F#类型提供程序提供了这种矛盾的解决方案，使数据（或者语言，例如我们的例子中调用了 R 语言）可以在阻力很小的情况下使用，且可以安全的方式发现，具备静态类型的所有好处。

F#在机器学习这个特殊领域中的另一个与众不同的优点是可同时用于探索和生产的能力。我们在本书中主要聚焦于第一个方面，在快速反馈循环中探索数据，逐渐改进模型。但是，一旦思路稳定，将代码从脚本提升为模块或者类，并将其转换为成熟的程序库都相当简单，这在前几章中都得到了说明。你可以预期，得到和.NET 语言中的正常性能相同的表现——也就是说，相当不错。而且，到那时候，可以在生产环境中运行代码，与.NET 代码库集成，不管那些代码是 C#、VB.NET 还是 F#。从探索到生产都使用同一种语言有真正的价

值。我曾经在许多地方看到这样的开发过程：研究团队用一组工具及语言创建模型，并将其传递给开发团队，由他们选择重写所有程序（以及引起的所有问题），或者尽其所能将外来的工具集成到生产系统中运行。F#可以为这种矛盾提供有趣的解决方案，可以同时作为研究用的探索语言和开发人员所用的生产就绪语言。

9.4 下一步是什么？

那么，现在你已经是机器学习的专家了吗？如果答案令人失望，我感到很遗憾，但是本书只是浅尝辄止，还有许多知识需要学习。尽管如此，如果你喜欢这个主题，好消息是，供你学习的有趣材料无穷无尽（作为出发点，我推荐 Coursera 的 Andrew Ng 所开的课程，并尝试参加 Kaggle 的一些竞赛）。机器学习的发展很快，这也是我如此热爱这个领域的原因。

更重要的是，虽然你可能还不是专家——但是这方面的专家本来就不多，因为机器学习的主题太大了。然而，你现在可能已经比大部分软件工程师更了解这一主题，应该可以将我们一起讨论的思路高效地用到自己的项目中了。最重要的是，我希望能够说服你，机器学习并不像第一眼看到的那么复杂，它充满了对数学家和软件工程师同样有趣的问题，也充满了乐趣。所以，亲手尝试，做些了不起的事，并享受快乐吧！

Apress®

专业人士写给专业人士的书

机器学习项目开发实战

本书向读者展示了，如何利用简单的算法和技术，从数据中学习，构建更聪明的.NET应用，以解决现实世界中更广泛的问题。读者将在熟悉的Visual Studio集成开发环境中，使用.NET平台中最适合于机器学习的F#语言开发机器学习项目。如果你对F#还很陌生，本书将教会你入门所需的知识；如果你对F#比较熟悉，本书将是你在机器学习领域实践该语言的新的机会。

在一系列令人着迷的项目中，读者将学到：

- 从头开始构建一个光学字符识别（OCR）系统；
- 编写一个通过例子学习的垃圾邮件过滤器；
- 使用F#强大的类型提供程序与外部资源接口（在本书中是来自R语言的数据分析工具）；
- 将数据转换为信息量更大的特征，并用它们作出精准的预测；
- 在不知道目标的情况下找出数据中的模式；
- 用回归模型预测数值；
- 实现一个可以从经验中学习玩法的智能游戏。

在阅读本书的过程中，你将学到适用于解决各种现实问题的基本思路，包括从广告到金融、医药和科学研究的多个领域。虽然有些机器学习算法使用了高级的数学理论，但是本书的焦点是简单而高效的方法。如果你喜欢挖掘代码与数据，那么这本书就是为你所编写的。

异步社区 www.epubit.com.cn
新浪微博 @人邮异步社区
投稿/反馈邮箱 contact@epubit.com.cn

美术编辑：董志桢

分类建议：计算机／人工智能
人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-42951-3

ISBN 978-7-115-42951-3

定价：59.00 元